UNCLASSIFIED
URITY CLASSIFICATION OF THIS PAGE

A 217 670

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION George Washington University 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Department of Electrical Engineering and Computer Science Washington, DC 20052	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION U.S. Army Medical Research & Development Command 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER Contract No. DAMD17-84-C-4129
8c. ADDRESS (City, State, and ZIP Code) Fort Detrick Frederick, Maryland 21701-5012	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 63764A; PROJECT NO. 3M46 3764D995; TASK NO. AB; WORK UNIT ACCESSION NO. 017

DTIC ELECTE JAN 25 1988 S E

11. TITLE (Include Security Classification)
Pattern Recognition of Cardiovascular and Psychomotor Variability in Response to Pharmacological Agents

12. PERSONAL AUTHOR(S)
Murray H. Loew, Linda Sibert

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Annual Report	FROM 4/15/85 TO 4/14/86	1986 August	127

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
09	02		HRV; Pattern Recognition; Signal Processing; Time Series; Point Processing
06	16		

19. ABSTRACT (Continue on reverse if necessary and identify by block number) The goal of this project is the development of pattern-recognition and signal-processing methods that will provide indices of responsivity to challenge when applied to Army-supplied human cardio-vascular and psychomotor data. Time-series and point-process techniques will form the basis of the approach, and the assumptions that underlie the methods will be examined and tested. The relationship of infrequent and brief events, if any, to the indices will be elucidated.

This report presents the results of the work over the past year, which has proceeded along three parallel lines: the design, implementation, and testing of data-preprocessing steps that restore physiologic integrity to noise-corrupted data; the preliminary implementation and evaluation of several clustering and pattern-recognition methods; and the selection of a data-segmentation algorithm for the partitioning of time-series data. The work followed naturally from that of the previous year, in which we reviewed the state of the art of the understanding of the links between the noninvasive measurements described here, and the underlying physiology.

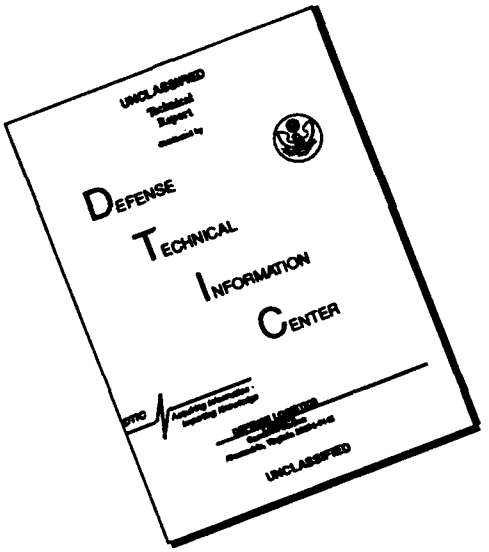
Plans are described for the third year of the work, which will combine those separate

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Mrs. Virginia Miller	22b. TELEPHONE (Include Area Code) 301/663-7325	22c. OFFICE SYMBOL SGRD-RMI-S

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

00 01 24 012

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

AD_____

Pattern Recognition of Cardiovascular and Psychomotor Variability in Response to Pharmacological Agents

Annual Progress Report

Murray H. Loew
Linda Sibert

August 1986

Supported by
UNITED STATES ARMY MEDICAL RESEARCH AND DEVELOPMENT COMMAND
Fort Detrick, Frederick, Maryland 21701-5012

Contract No. DAMD17-84-C-4129

Department of Electrical Engineering and Computer Science
The George Washington University
Washington, DC 20052

DOD Distribution Statement

TABLE OF CONTENTS PAGE

LIST OF FIGURES

LIST OF TABLES

Abstract

The goal of this project is the development of pattern-recognition and signal-processing methods that will provide indices of responsivity to challenge when applied to Army-supplied human cardiovascular and psychomotor data. Time-series and point-process techniques will form the basis of the approach, and the assumptions that underlie the methods will be examined and tested. The relationship of frequent and brief events, if any, to the indices will be elucidated.

This report presents the results of the work over the past year, which has proceeded along three parallel lines: the design, implementation, and testing of data-preprocessing steps that restore physiologic integrity to noise-corrupted data; the preliminary implementation and evaluation of several clustering and pattern-recognition methods; and the selection of a data-segmentation algorithm for the partitioning of time-series data. The work followed naturally from that of the previous year, in which we reviewed the state of the art of the understanding of the links between the noninvasive measurements described here, and the underlying physiology.

Plans are described for the third year of the work, which will combine those separate tasks into a single tool for physiologic state characterization.

FOREWORD

Citations of commercial organizations and trade names in this report do not constitute an official Department of the Army endorsement or approval of the products or services of these organizations.

1. Introduction

The study of heart-rate variability (HRV) has become of increasing interest, especially in the 10 years since the Biological Engineering Society held a meeting on the subject in London. In addition to the studies dealing with underlying physiology, analysis techniques, and applications to physiology, there has been work in applications; e.g., the estimation of levels of workload, and detection of mental illness, using the HRV.

In a monitoring environment, where an individual's ability to perform a task is to be described, it is important to have a rapid, unambiguous measure. Because of the relationship of sleep and sleep deprivation to performance, it seems reasonable to evaluate the effect of adding a sleep-related parameter to any kind of noninvasive measurement system.

Accordingly, we are considering in this work the combination of HRV and an activity measure (recorded with an actigraph or actometer) to assess performance. The ways in which the data are processed and described are presented in the following report.

2. Scope of Work and Progress to Date

Our approach makes use of pattern-recognition and signal processing techniques in the development of methods for classifying human cardiovascular and psychomotor response to challenge. Table 1 presents the sequence of tasks which constitute our effort. Detailed information about the topics represented by each box make up the bulk of this report.

It is important to note that a number of the tasks have been proceeding in parallel. Tasks 2.2, 2.4 (development of computer programs), 2.8 and 2.9 (development of programs and performance evaluation using sample data from outside this project) are essentially complete. The programs developed there are now ready for immediate application once Task 2.3 (segmentation of signals) has been completed.

2.1 Data

The actigraph and R-R interval signals supplied by the Army were each broken into 10 twenty-four hour periods which are further subdivided into half-hour files. The data are binary with no end-of-file marks. These files have been converted into ASCII files, uploaded onto the IBM 4341, and stored on a tape. A VAX 11/780 version of that tape also has been produced.

The data come from a study of the effect of different doses of atropine on the heart and physical movement. Two males in their twenties participated. The subjects were hospitalized, ambulatory but restricted. During the experiment, an ECG and an actigraph signal were continuously recorded. The subjects wore actigraphs on their right wrists which recorded a voltage signal proportional to the amount of g-force of the wrist in the lateral-medial plane.

TABLE 1

TASK SEQUENCE

2.1	Receive raw R-R interval and actigraph waveforms from the Army.
2.2	Remove noise. Base algorithms on physiological criteria and known instrumentation artifacts. Collect statistics about number and kinds of problems.
2.3	Segment actigraph and R-R interval signals.
2.4	Extract features from each segment. Work in both the time domain and frequrncy domain. Study correlation among features. Choose those least correlated.
2.5	Use clustering algorithms to learn the natural groupings of the segments in feature space.
2.6	Use segment labels to be provided by Walter Reed to evaluate the results of clustering.
2.7	Reduce the number of features using dimensionality-reduction techniques.
2.8	Build classifiers. Examine both parametric and nonparametric models.
2.9	Test classifiers with new data and evaluate performance.

The experiment was conducted over 5 non-contiguous days, one for each level of atropine. Atropine was given about 30 minutes after the start of each session, either in an intravenous or intramuscular form. Table 2 outlines the protocol used.

Table 2

EXPERIMENT PROTOCOL

Study Schedule	Subject 1		Subject 2	
Day	Start Time	Dose	Start Time	Dose
1	0907	0.0	0818	1.0 mg IM
2	0717	0.5 mg IM	0847	2.0 mg IV
3	0835	1.0 mg IM	0921	0.5 mg IM
4	0837	2.0 mg IM	0912	0.0
5	0835	2.0 mg IV	0909	2.0 mg Im

The Army performed the following operations on the data before providing it for analysis. Both the ECG and actigraphy signals were converted to digital form. On playback, the actigraphy signal was amplified (Oxford Event Demodulator Amplifier model PM-3) and filtered (bandpass filter 1.04-4 Hz) by Coulbourn Instrument model S75-36). The signal was sampled with a programmable digital oscilloscope (Norland model 3001A with 128K-word buffer memory for the channel). The signal was sampled at an effective rate of 15 Hz and the RMS value of each 2-sec interval stored in a Corona computer running under MS-DOS 1.00. For the ECG signal, the R-peaks were detected using a maximum-slope detection algorithm with a real sampling rate of 400 Hz. The R-R intervals were stored. Both the processed ECG and actigraphy data were divided into half-hour files.

2.2 Preprocessing

Before the data can be analyzed, any noise produced by instrumentation and physiological artifacts must be removed. Data adjustment algorithms have been written to process R-R interval and actigraph data. Statistics on the number and kinds of problems which appear in the data will be kept for future analysis which may lead to more streamlined data-adjustment algorithms.

2.2.1 R-R Interval Data

Noise can be introduced into the ECG data in three ways: (1) physiologic artifacts, (2) tape-drive-induced artifact, and (3) residual instrument noise. To identify the noise in the signal, we determined an acceptable range for heart rate. We reviewed the physiological literature which indicated that a normal heart rarely falls below a resting heart rate of 35 beats per minute (bpm) or above 200 bpm which can be reached during extremely vigorous exercise. A

reasonable acceptable range, therefore, is 40 bpm to 180 bpm (corresponding to R-R intervals between 333 ms and 1499 ms). By this standard, the ECG data are fairly noise-free. On average, only 2% of the intervals in each 30-minute file are out of range.

A three-part algorithm, described in detail below, was developed to eliminate the noise from each ECG file. The first part truncates the data file to eliminate the tape stoppage noise which appears as a cluster of intervals less than 333 ms, about 150 intervals from the end of each file. The second part makes adjustments to the data in a conservative manner. The algorithm is designed so that most of the correction happens here. The third part is made up of two sections, both of which are designed to change the data file just enough to allow the second part of the algorithm to resume. The flow chart of the following algorithm can be found in Appendix 1.

At the beginning and end of this process, information about the original data set, the difficulty of the noise-elimination task, and the resulting noise-free data set is stored for use in developing confidence measures needed in future analysis. Specifically, the following information about each file is kept:

1. the amount of time truncated from the data file which represents bad data due to tape stoppage.

2. the number of times the algorithm reached a point between the second and third parts, a measure of how difficult it was to correct the data.

3. the number of intervals less than 333 ms.

4. the number of intervals between 1500 and 2999 ms.

5. the number of intervals greater than 2999 msec..

6. the number and lengths of runs of intervals between 333 and 1499 ms, our acceptable range.

After truncation, the majority of out-of-range intervals in each file fall between 1500 and 2999 ms. Those greater than 2999 ms constitute the next largest group. Intervals less than 333 ms rarely occur.

2.2.1.1 Algorithm Part 1: Preprocessing and Elimination of Noise Caused by Stopping the Tape

The goal of this part of the algorithm is to cut the data file at the earliest interval in the cluster of short intervals caused by stopping the tape during processing. The method used is a search procedure which locates this point by using our knowledge that a cluster of short intervals, occuring more densely than anywhere else in the file, occurs between 100 and 200 intervals from the end of each file.

First, five non-overlapping windows, which each hold five intervals, move as one window from the end of each file toward the

beginning, examining one interval at a time. When three out of the five windows contain at least one interval less than 333 ms, the search stops. We want our chosen interval to be in a region containing many short intervals and not to be an isolated point. The interval less than 333 ms which is closest to the end of the file is selected. From that interval, we jump 150 intervals back in time into a section of the file which should be near the cluster of short intervals. At this point, 2 windows of 15 intervals each are created. They are moved back in time, in a step-wise, non-overlapping fashion, until no short intervals are contained in the left window. When this occurs, the focus switches to the right window which is moved one interval at a time toward the end of the file (i.e., to the right), until the window contains at least three intervals smaller than 333 ms. The file is truncated at this earliest short interval.

2.2.1.2 Algorithm Part 2: Conservative Approach

For the remaining discussion of the algorithm, it is useful to visualize an interval, with a left neighborhood made up of the intervals which precede it in time, and a right neighborhood that contains the intervals which follow it in time. These neighborhoods will always be qualified by a number which is the size of the neighborhood (i.e., the number of members in the set called neighborhood).

In Part 2, the out-of-range data values (those not between 333 and 1499 ms) are broken into three cases. The first is called short and contains all values less than 333 ms. The second is called medium and contains the intervals between 1500 and 2999 ms. The large case contains all intervals greater than 2999 ms.

Short intervals can be the result of equipment-caused noise or an extra systole which is a premature contraction of the heart originating at a site other than the usual pacemaker. An extra systole can cause the cardiac cycle to lengthen slightly. In either case, a false R-peak has been detected. This short interval probably belongs to one of its immediate neighbors (if it has been caused by noise), and its sum with the neighbor should be close to the lengths of the other intervals surrounding it. If it is an extra systole, the length of the sum will be somewhat larger than that of those intervals surrounding it.

Case 1, which deals with these short intervals, requires that three intervals in the acceptable range lie on either side of the short interval. This short interval is than added to each neighbor, and these two sums are compared with the mean of the six closest neighbors. The sum closer to the mean is chosen and accepted as the correct interval, if it is less than 1500 ms.

Medium intervals are the result of system noise. The average heart interval is about 750 ms, and 2999 ms, the longest medium interval, represents four 750-ms intervals. Our strategy, therefore, is based on the fact that these intervals represent no more than three missed peaks.

In Case 2, each medium interval is divided into equal sub-intervals based on the size of the median interval of its surrounding neighborhood. To qualify for this procedure, a medium interval must have either four contiguous neighbors on each side in the range of 333

to 1499 ms. or three contiguous neighbors on each side in that range. The median of those neighbors is calculated.

We chose not to use an unequal number of in-range neighbors from each side in order not to bias the median determination. Four and three were chosen since they represent enough time to estimate the rate in that area. Including more neighbors would give influence to intervals too far away in time to be related. It has been noted [1,2] that heart rate can be altered by the sympathetic system within a few beats at most and within a cycle at best.

The interval is partitioned into sub-intervals the size of the neighborhood median: any remaining time is distributed among the sub-intervals. At this point, if any interval is greater than 1499 ms or the remainder is greater than half the mediam, the original interval is repartitioned into the number of previous subintervals plus one, and the new remainder distributed.

Intervals exceeding 2999 ms arise when three or more consecutive R-peaks are missed; we call this situation the large case. Any information about heart rate acceleration or deceleration would be lost if this larger span of time simply were divided into equal pieces. Another strategy has been chosen.

Again, two contiguous neighborhoods of intervals in the acceptable range are required, but this time each neighborhood is a "spanning set" made up of a varying number of intervals whose total time equals or just exceeds the amount of time in the large interval. From these spanning sets, we calculate a left limit and a right limit for an arithmetic progression that is used to divide the large interval. When the variation in a spanning set is in the range ±20%, we use the mean of the set as the limit associated with that spanning set. Otherwise, we use the median, which is less affected by extreme values.

2.2.1.3 Algorithm Part 3

The program iterates through Part 2 until there are no more out-of-range intervals which meet the requirements, principally that an interval must have intervals in the range of 333 to 1499 ms on either side of it. At this point, Section A of Part 3 is invoked. All intervals which meet its criteria are now adjusted. Again, the intervals are identified as short, medium, and large but medium and large are handled in the same way. The medium and large intervals are merged into the class medium/large.

Both cases in Section A use an alternate approach from that used in Part 2 which required intervals in the acceptable range to be on both sides. Here, only a contiguous string of 10 in-range intervals which lie on one side of the out-of-range interval is required. Ten intervals was chosen because an interval much farther away would add little information about the true nature of the heart rate, while the fewer might contain too little information about how the heart rate is changing. The areas in which we are now working have clusters of out-of-range intervals (there are only acceptable intervals on one side), implying considerable noise.

For the short case, there could be a run of ten intervals in the acceptable range on either side if, in Part 2, the interval when added

to a neighbor was greater than 1499 ms. Therefore, it is necessary to check both sides of the out-of-range interval. The short interval is added to either nearest neighbor which meets the criterion of being in a run of ten and, if more than one, the new value closer to the mean of its associated run of ten is chosen as the correct interval. Otherwise, if only one run of ten was found, the sum of the interval and its near neighbor is accepted. This new interval must be less than 1500 ms.

The medium/large case is a variation of the large case in Part 2. Part 2 bridges the gap created by the out-of-range value by filling in values with an arithmetic progression run from the mean of the left neighborhood to that of the right. In the medium/large case, however, since we have already identified a run of ten intervals in the acceptable range on one side, we have one of those neighborhood statistics. What we do not have is something for the other end of the progression.

Our solution works only with the five intervals closest to the out-of-range interval, among the run of 10. One limit is the median of that neighborhood of five. It serves as one end of the arithmetic progression. The other end of the progression is the average of: (1) twice the mean of the neighborhood of five; (2) the mean of whatever intervals in the acceptable range are found within five intervals on the other side. Whatever close, in-range intervals exist on the other side should have some limited influence on the nature of the arithmetic progression. After the arithmetic progression is calculated, the remaining time is distributed uniformly.

If no out-of-range intervals could be changed in Section A, Section B is used because something has to be altered to allow Part 2 to resume. Hence, only one or two out-of-range intervals are changed. The two cases are again short and medium/large.

First, the longest run of in-range intervals in the entire file is identified. The medium/large out-of-range values at either end are chosen first. The medium/large case is similar to that found in Section A. One end of the arithmetic progression is the median of the five closest in-range values or however many in-range values there are in the run. The other end is the mean of that five (or howevermany in-range values there are in the run), modified by any in-range values lying within five of the medium/large value. Any remainder after the progression has been calculated is uniformly distributed.

A short interval is adjusted only if nothing has been changed by the Section B medium/large algorithm, because this procedure is the most arbitrary. The short interval is simply added to its smaller neighbor and no test for variability or size is made.

2.2.2 Noise Removal: Actigraph Data

Each data file has an offset which must be subtracted to produce a zero-minimum signal. The signals are otherwise quite clean. These are true time-series data, of fixed length per file.

2.3 Segmentation

Once the noise has been removed, we will subdivide the signals into pieces at points where the nature of the signal changes. Since each 24-hour R-R interval signal has a companion 24-hour actigraph signal, several approaches are available. First, the R-R interval signal and the actigraph signal may be segmented separately and the correspondence of their boundaries examined.

A second approach is a hierarchical one in which the segmentation of one signal would determine the boundaries of the other. The hierarchical technique will be examined during the next year. The actigraph signal will guide the process because it represents only activity, a simpler physiological event than heart rate which contains many components like respiratory sinus arrhythmia (RSA) and biorhythms. Statistics on the number and lengths of segments will be kept to aid in the evaluation of the method.

The segmentation algorithm we have chosen [3] looks for differences in the parameters between two segments of the signal which have been modeled as autoregressive (AR) processes of the same order, and fixes a boundary between dissimilar pieces.

The approach is to model the finite-duration random time series by a stationary, normally-distributed autoregressive process of order p. Stationarity means, qualitatively, that the graph of the time series looks about the same near one time as near another. More formally, all statistical properties of stationary time series remain unchanged when the period of observation is shifted forward or backward in time. In particular, the mean and the variance (as well as the higher-order moments) do not change with time, and the autocovariance between two values separated by τ time units depends only on τ. Although many real time series may not fulfill those conditions perfectly, the tools that are derived under the assumption of stationarity often work quite well on those series.

By autoregressive of order p, we mean that the series $\{r(t)\}$, $t=1,2,\ldots,N$ can be written

$$\sum_{i=0}^{p} a_i\, r(t-i) = \sigma\, u(t)$$

where $a_0=1$ and $a_1, a_2, \ldots, a_p$ are coefficients that allow $r(t)$ to be expressed in terms of the p previous values of the series. The error, or disturbance, term $\sigma u(t)$ is assumed to be an uncorrelated stationary normally-distributed series with variance σ^2.

Standard methods [4] exist for estimating the parameters a_i, and if two time series $r(t)$ and $s(t)$ exist, it is possible to compute the joint likelihood (probability) of $\{r(t)\}$ and $\{s(t)\}$ conditioned on the first p observations of each sequence. We thus can find the maximum likelihood under the null hypothesis that the parameter sets are equal and compare it to the maximum likelihood under the alternative hypothesis that the parameters are arbitrary. A maximum-likelihood

ratio then can be found that can be transformed into a distance measure d such that d=0 for sequences having identical parameter sets. This measure is derived in [3] as follows:

The joint likelihood of {r(t)} and {s(t)} is

$$\mathcal{L} = (\sigma_R\sqrt{2\pi})^{-N'_R} (\sigma_S\sqrt{2\pi})^{-N'_S} \exp\left\{ - \frac{N'_R}{2\sigma_R^2} \underline{a}_R^T C_R \underline{a}_R - \frac{N'_S}{2\sigma_S^2} \underline{a}_S^T C_S \underline{a}_S \right\}$$

where

N_R = length of {r(t)}
N_S = length of {s(t)}
p = order of process
$N'_R = N_R - p$
$N'_S = N_S - p$

C_R and C_S are the covariance matrices of {r(t)} and {s(t)}, respectively

$\underline{a} = (a_0 a_1 a_2 \ldots a_p)$

Let $\mathcal{L}$ denote the maximum likelihood under the null hypothesis that $\underline{a}_R = \underline{a}_S$ and $\sigma_R = \sigma_S$. Thus $\mathcal{L}_0$ can be written as

$$\mathcal{L}_0 = (\sigma_p\sqrt{2\pi})^{-(N'_R + N'_S)} \exp\left\{ - \frac{1}{2} (N'_R + N'_S) \right\}$$

where $\sigma = \underline{a}^T C \underline{a}$ and

σ_p = pooled estimate of r.

Similarly let $\mathcal{L}_1$ denote the maximum likelihood under arbitrary parameters settings. $\mathcal{L}_1$ is written as:

$$\mathcal{L}_1 = (\sigma_R\sqrt{2\pi})^{-N'_R} (\sigma_S\sqrt{2\pi})^{-N'_S} \exp\left\{ - \frac{1}{2} (N'_R + N'_S) \right\}$$

The maximum likelihood ratio is given by

$$\lambda = \frac{\mathcal{L}_0}{\mathcal{L}_1} = \sigma_p^{-(N'_R + N'_S)} \cdot \sigma_R^{N'_R} \cdot \sigma_S^{N'_S}$$

Detine d as

$$d = -2 \ln \lambda$$

In general

$$d = (N_R' + N_S') \ln \sigma_p^2 - \left(N_R' \ln \sigma_R^2 + N_S' \ln \sigma_S^2 \right) \qquad (1)$$

with σ^2 and C being computed separately for R,S, and P.

Thus d is a measure of the statistical differences between the two signal segments. The larger d is, the more the parameters of the segments are expected to be different. Because a logarithm was used in the derivation of d to make it zero for identical sets, it is called an "entropy distance" by Chen [3].

The order p of the autoregressive model, the size of the window w, and the threshold d_{th}, above which the segments are considered different, must all be estimated. A suggestion for estimating the threshold d_{th} is to construct a histogram of the entropy distances which have been calculated for all adjacent pairs of segments of the signal after it has been initially segmented into equal-length pieces, each of size w. They observe that such a histogram seems to have Chi-square distribution which might help in selecting a cut-off. It is noted [5] that $w \geq (p/3)^2$ where p is the order of the underlying AR process, because "...in any ergodic time series where statistical parameters are calculated as time averages a minimum interval of length L is necessary to estimate the statistical parameters with sufficient accuracy" [p.31]. In general, however, selecting p,w, and d_{th} requires experience and a general understanding of the characteristics of the signal.

Our segmentation algorithm consists of a broad initial search for an optimum boundary followed by a specific point-by-point search. These two parts, themselves, are each broken down into two subsections. Part 1 begins with Rough Boundary Searching, followed by Optimum Boundary Searching. The final search, carried out in Part 2, takes one of two forms. The choice is based on the location of the currently selected optimum boundary point.

2.3.1 Part 1

2.3.1.1 Rough Boundary Searching

Once w, the window length, and p, the order of the autoregressive process, have been selected the signal is partitioned into w-length segments, each labeled as in Figure 1. The last point in each segment s_i is called a node and labeled n_i. For consistency, the first segment begins at point 2 of the signal, and point 1 is labeled n_0.

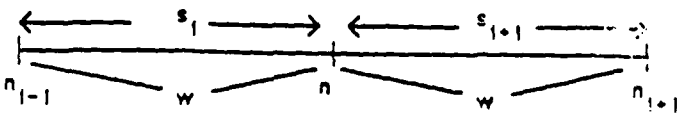


Figure 1

This crude segmentation of the signal is refined in the remaining sections of the algorithm. It is important to note that the number of segment boundaries, which is initially a function of the window length w, is a fixed upper limit. The number of boundaries can only decrease.

The entropy distance between each pair of adjacent segments s_i and s_{i+1} is calculated using Eq.(1), and that value, labeled $d(s_i,s_{i+1})$, is associated with the intervening node n_i.

An entropy distance threshold d_{th} is calculated for the data. This threshold is used during Optimum Boundary Searching.

For the remainder of the discussion of the segmentation algorithm, the first point in the data file will be referred to as the leftmost point and the last as the rightmost point.

2.3.1.2 Optimum Boundary Searching

Locate the rightmost $d(s_i,s_{i+1})$ which is greater than d_{th}, the threshold distance. All the n_i associated with each $d(s_i,s_{i+1}) < d_{th}$ are no longer boundaries because the segments s_i and s_{i+1} are considered statistically similar since their entropy distances are below the threshold. If no $d(s_i,s_{i+1})$ is greater than d_{th}, then the chosen threshold should be re-evaluated.

The rightmost $d(s_i,s_{i+1}) \geq d_{th}$ is associated with node n_i and is called the current optimum boundary between the two segments. When the characteristics of the signal change between any two consecutive segments, the entropy distance between those segments will exceed the threshold. A finer search is then conducted in s_{i+1} for a better optimum boundary. The left segment s_i is not searched at this time. It will be searched if the final optimum boundary is to the left of s_{i+1} after the completion of Part 2 of the algorithm. We divide s_{i+1} into equal subsegments of length w_s. The nodes are labeled in the same manner as before: Node n_i is relabeled m_0 and the following points are $m_1,m_2,\ldots,m_n$. The last subsegment m_n is also labeled n_{i+1}. The labeling of this section of the signal is shown in Figure 2.

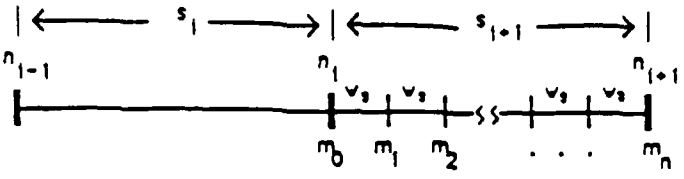


Figure 2

Two pairs of windows (s'_i, s'_{i+1}) and (s''_i, s''_{i+1}) are created. (Initially, $s'_i = s_i$.) They are shown in Figure 3. We call s'_i and s''_i test windows, and s'_{i+1} and s''_{i+1} reference windows [5]. A window must be at least w points long.

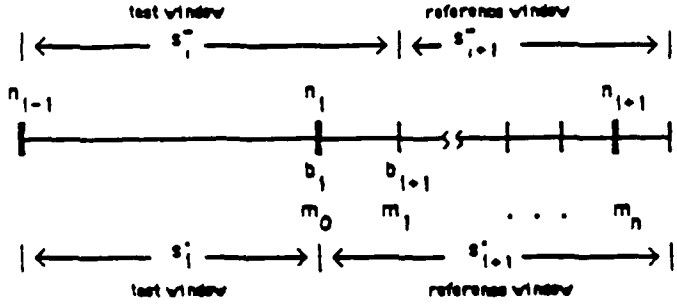


Figure 3

These windows move through s_{i+1} in w_s increments in a search for a better optimum boundary than the currently selected one. The process will stop after m_{n-1} is considered. Stopping at m_{n-1} rather than at m_n is a departure from Chen's specification. It was done to simplify the point-by-point search carried out in Part 2. The sizes of the windows will vary over time. The only conditions are (1) that both test windows always start at n_{i-1}, (2) that s''_{i+1} is always w long (the minimum window length), and (3) that both reference windows always end at the same point.

At the start, the windows are positioned as in Figure 3. Test windows s'_i and s''_i begin at n_{i-1}. These test windows end at the same point at which their reference windows begin; namely s'_{i+1} at m_0 and s''_{i+1} on m_1. The junction of both s' windows is always the current optimum boundary and is labeled b_i. Therefore, $m_0 = b_i$. The junction of both s" windows is always labeled b_{i+1}. Therefore, $m_1 = b_{i+1}$. Both reference windows end at the point $b_{i+1} + w$.

The entropy distance between the s' windows is calculated again using Eq.(1). The value is associated with b_i. The entropy distance between the s" windows is likewise calculated and associated with b_{i+1}. The point associated with the larger entropy distance is chosen as the current optimum boundary. If the distances are the same then the current optimum boundary is not changed.

The end of s_i is now moved to the current optimum boundary and the end of s_i'' is moved a distance w_s from its former position. The label b_i again is associated with the optimum boundary and the end of s_i. The end of s_i'' is associated with b_{i+1}. It is necessary to check that s_{i+1} and s_{i+1}'' both end at $b_{i+1} + w$.

A new pair of entropy distances is calculated, the larger selected, and the windows moved. The procedure is followed over all subsegments up to and including m_{n-1}. The final optimum boundary is located at b_i.

2.3.2 Part 2

The search continues to the left and right of b_i in a point-by-point manner. The range to be covered is $(b_i - w_s, b_i + w_s)$, and the search is carried out in the same manner as before. The point $b_i - w_s$ is considered the current optimum bondary. The only difference is that when s_i'' changes position, it moves up only one point. Figure 4 shows the arrangement of the windows.

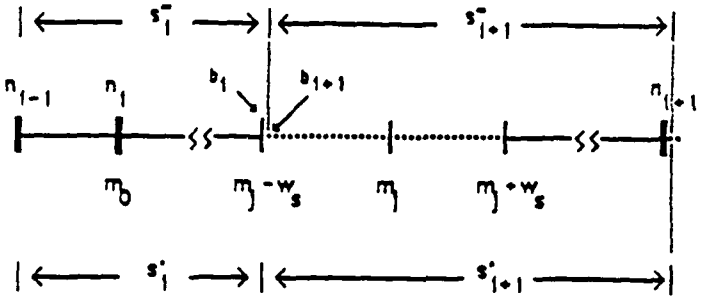


Figure 4

After all points are examined and if $b_i \neq m_0 \neq n_n$, then the point associated with b_i is accepted as the optimum boundary. However, if $b_i = m_0 = n_n$, then the point at b_i is accepted only if the final b_i is either at m_0 or to the right of m_0. If b_i is to the left of m_0, then it is necessary to look at the entropy distance associated with the previous pair of windows $d(s_{i-1}, s_i)$. If $d(s_{i-1}, s_i) \geq d_{th}$ then no boundary is accepted. If $d(s_{i-1}, s_i) < d_{th}$, then we consider $d(s_{i-1}, s_i)$ to be greater than d_{th} and search for an optimum boundary in those segmenets.

The algorithm is repeated for all $d(s_i, s_{i+1}) \geq d_{th}$.

The limitations of this algorithm are minor. First, the data file initially must be segmented into equal pieces. Rarely will this division come out equally, so some points at the end of the file will be lost. Second, the first segment will not be searched because only s_{i+1} is considered, and similarly, the last segment will never be searched because of the requirement that s_{i+1} must be v points long. However, these lost points represent only a small fraction of total in the file.

2.4 Feature Extraction

Features will be extracted from each segment. The goal of feature extraction is the characterization of each segment by a set of measurements (the *feature vector*) that are invariant in the presence of noise or sample-to-sample differences. Ideally, the feature vectors will be similar for segments of the same type, and easily distinguished from each other for dissimilar segments. The notion of feature extraction from a sample may be thought of conveniently as the location of a point in an n-dimensional space. The coordinates of the point are the values of the n features, and the label of the point is its class. Thus, we intend that non-identical segments of the same class will have representations which are close to each other. The pattern recognition problem then is equivalent to the construction of boundaries in the n-dimensional space that result in the separation of groups of points on one label from groups of points with other labels. Ideally, each group would contain points all having the same label.

Our candidate feaures will include time-domain measurements (e.g., the number of zero-crossings, amplitude-based measures (mean-square, other moments, histogram shape), inter-peak intervals and slopes, autoregressive (AR) parameters, and frequency-domain measurements (e.g., shape of the power spectrum: location of maximum-power band, number of maxima).

2.5 Clustering

Clustering seeks to partition a given data set into homogeneous subsets (clusters) by considering similarities of data points (feature vectors) in each subset and their relationship to the elements of other subsets. Typical similarity measures are: the Euclidean distance, the city-block distance, the Minkowski metric (a generalized Euclidean distance), and a quadratic distance function [6]. The use of such metrics as similarity measures can be justified by the heuristic argument made above that points in the same cluster should be close to each other and, at the same time, distant from the elements of other clusters.

There are basically two approaches to clustering. The first, known as the *dynamic clustering method*, uses an iterative algorithm to optimize a clustering criterion function. Various criteria of clustering have been suggested in the literature. Among these, the most useful have proved to be the family of functions that quantify the average affinity of data points to cluster representatives. At each iteration of a dynamic clustering algorithm, data points are assigned to clusters, the number of which must be specified in advance. The assignment is performed on the basis of the points' similarity with the current cluster representatives. In subsequent steps, the cluster representatives are updated to reflect any changes in the data-point assignments. Those new cluster models are used in the next iteration to reclassify the data, and the process is continued until a stable partition is obtained.

A second approach, known as *hierarchical clustering*, is non-iterative. At any stage of a hierarchical clustering algorithm the two most-similar existing clusters are merged, thus reducing the number of potential clusters by one. After n-1 steps where n is the cardinality of the set being analyzed, the algorithm terminates. The number of clusters in the data set need not be known *a priori*. Rather, natural clusters of points in the data set, for a given measure of similarity, are detected by assessing the changes in the values of the measure at various stages of the algorithm.

A number of very good algorithms of both types are documented in the literature [7,8]. These algorithms will be evaluated using the features extracted as indicated above.

2.6 Evaluation

At this stage of the work, the Army will provide the physiological-state labels, and the times at which they begin in each of the ten 24-hour signals. This information will permit evaluation of the segmentation method and of the several clustering methods. If the algorithms are working well, then (1) the partitions found in this work will agree with the beginning-points of the states provided by the Army, and (2) most of the points in a given cluster will have the same label.

2.7 Feature Selection

The features chosen will be prewhitened both approximately (by removing all but one of the features making up a set of highly-correlated features) and exactly (by a diagonalization of the features' correlation matrix). The resulting sets will be evaluated in several ways:

(1) by the probability of error associated with their use in a classifier (see below)

(2) by the Karhunen-Loeve transformation (which uses the eigenvalues of the features' covariance matrix to rank them according to their intrinsic ability to separate the samples); and

(3) by the homogeneity of the clusters found in Sec. 2.5.

2.8 Classifier Design

2.8.1 Parametric Methods

2.8.1.1 Introduction

The purpose of pattern recognition is to determine to which category or class a given sample belongs. Feature extraction provides a set of numbers which make up the observation vector. The observation vector serves as the input to a decision rule by which we assign the sample to one of the given classes. Let us assume that the observation vector is a random vector whose conditional density

function depends on its class. If the conditional density function for each class is known (this is the parametric case), then the pattern recognition problem becomes a problem in statistical hypothesis testing.

Here we discuss the two-class problem, which arises because each sample belongs to one of two classes, ω_1 or ω_2. The conditional density functions and the *a priori* probabilities are assumed to be known. See Appendix F for variable definitions.

2.8.1.2 The Bayes Decision Rule for Minimum Error

Let X be a feature vector, and let it be our purpose to determine whether X belongs to ω_1 or ω_2. A decision rule based simply on probabilities may be written as follows:

$$P(\omega_1/X) \gtrless P(\omega_2/X) \rightarrow X \in \begin{cases} \omega_1 \\ \omega_2 \end{cases} \qquad (2.8.1\text{-}1)$$

The *a posteriori* probabilities $P(\omega_i/X)$ may be calculated from the *a priori* probabilities $P(\omega_i)$ and the conditional density functions $p(X/\omega_i)$, using Bayes' theorem, that is

$$P(\omega_i/X) = \frac{p(X/\omega_i)\, P(\omega_i)}{p(X)}$$

Since $p(X)$ is common to both sides of the inequality (2.8.1-1), the decision rule of (2.8.1-1) can be expressed as

$$p(X/\omega_1)P(\omega_1) \gtrless p(X/\omega_2)P(\omega_2) \rightarrow X \in \begin{cases} \omega_1 \\ \omega_2 \end{cases}$$

or

$$l(X) = \frac{p(X/\omega_1)}{p(X/\omega_2)} \gtrless \frac{P(\omega_2)}{P(\omega_1)} \rightarrow X \in \begin{cases} \omega_1 \\ \omega_2 \end{cases} \qquad (2.8.1\text{-}2)$$

The term $l(X)$ is called the *likelihood ratio* (related to, but not the same as, that used in segmentation) and is the basic quantity in hypothesis testing. We call $P(\omega_2)/P(\omega_1)$ the *threshold value* of the likelihood ratio for the decision. Sometimes it is more convenient to write the minus-log-likelihood ratio rather than writing the likelihood ratio itself. In that case, the decision rule (2.8.1-2) becomes

$$-\ln l(X) = -\ln p(X/\omega_1) + \ln p(X/\omega_2) \lessgtr \ln \{P(\omega_1)/P(\omega_2)\} \rightarrow X \in \begin{cases} \omega_1 \\ \omega_2 \end{cases} \qquad (2.8.1\text{-}3)$$

The direction of the inequality is changed because we have used the negative logarithm.

Equation (2.8.1-2) or (2.8.1-3) is called the *Bayes test for minimum error.*

In general, the decision rule of (2.8.1-3), or any other decision rule, does not lead to perfect classification. In order to evaluate the performance of a decision rule, we must calculate the *probability of error*, that is, the probability that a sample is assigned to the wrong class.

Let I_1 and I_2 be the regions in the domain of X such that $P(\omega_1/X) > P(\omega_2/X)$ and $P(\omega_1/X) < P(\omega_2/X)$, respectively. Then, if $X \in I_i$, we assign the sample to class ω_i. The probability of error can be calculated as follows:

$$\varepsilon = Pr\{error\} = Pr\{error/\omega_1\}P(\omega_1) + Pr\{error/\omega_2\}P(\omega_2)$$

If the sample belongs to ω_1, an error occurs whenever $X \in I_2$, and, similarly, if the sample belongs to ω_2, an error occurs whenever $X \in I_1$. Thus,

$$\varepsilon = Pr\{X \in I_2/\omega_1\}P(\omega_1) + Pr\{X \in I_1/\omega_2\}P(\omega_2)$$

$$= P(\omega_1)\int_{I_2} p(X/\omega_1)dX + P(\omega_2)\int_{I_1} p(X/\omega_2)\,dX$$

$$= P(\omega_1)\varepsilon_1 + P(\omega_2)\varepsilon_2$$

We can distinguish two types of errors: one which results from misclassifying samples from ω_1 and the other which results from misclassifying samples from ω_2. The total error is a weighted sum of these errors.

The problem of calculating the probability of error is solved essentially by the integration of density functions in an n-dimensional space. Therefore, it is sometimes more convenient to integrate the density function of the likelihood ratio $p(l/\omega_i)$, which is one-dimensional.

$$\varepsilon_1 = \int_0^{P(\omega_2)/P(\omega_1)} p(l/\omega_1)dl \qquad (2.8.1\text{-}4)$$

$$\varepsilon_2 = \int_{P(\omega_2)/P(\omega_1)}^{+\infty} p(l/\omega_2)dl$$

where the region of integration of (2.8.1-4) is from 0 to $P(\omega_2)/(P\omega_1)$ because the likelihood ratio is always positive.

In the common case when the $p(X/\omega_i)$'s are normal with expected vectors M_i and covariance matrices Σ_i, the decision rule of (2.8.1-3) becomes

$$h(X) = -\ln L(X)$$

$$= \frac{1}{2}(X - M_1)^T \Sigma_1^{-1}(X - M_1) - \frac{1}{2}(X - M_2)^T \Sigma_2^{-1}(X - M_2) + \frac{1}{2}\ln\frac{|\Sigma_1|}{|\Sigma_2|}$$

$$\lessgtr \ln\frac{P(\omega_1)}{P(\omega_2)} \rightarrow X \in \begin{matrix} \omega_1 \\ \omega_2 \end{matrix} \qquad (2.8.1\text{-}5)$$

Equation (2.8.1-5) shows that the decision boundary is given by a quadratic form in X.

When $\Sigma_1 = \Sigma_2 = \Sigma$, the boundary becomes a linear function of x_i as

$$h(X) = (M_2 - M_1)^T \Sigma^{-1} X + \frac{1}{2}(M_1^T \Sigma^{-1} M_1 - M_2^T \Sigma^{-1} M_2)$$

$$\lessgtr \ln\frac{P(\omega_1)}{P(\omega_2)} \rightarrow \varepsilon \begin{matrix} \omega_1 \\ \omega_2 \end{matrix}$$

2.8.2 A nonparametric Method: The Empirical Processing Algorithm (EPA)

2.8.2.1 Origin of the EPA and the Structure of Its Classifier

2.8.2.1.1 Background and Development of the EPA

A nonparametric problem in two-class pattern recognition was considered by Henrichon [9] and by Loew and Fu [10]: one approach involved the use of the empirical distribution function as an approximation to the underlying distribution function. A principal result obtained there in the case of one-dimensional observations (i.e., the feature vector has only one element) was an algorithm for determining the relative extrema of the function:

$$\frac{f(x|\omega_1)}{f(x|\omega_2)} \quad \begin{matrix} \geq 1. & x \in \omega_1 \\ < 1. & x \in \omega_2 \end{matrix} \qquad (2.1)$$

Here $f(x|\omega_1)$ and $f(x|\omega_2)$ are the (assumed) continuous cumulative distibution functions (cdf's) for the populations (classes) ω_1 and ω_2.

The need to find the relative extrema of (2-1) was motivated by the following observations. If we assume equal *a priori* class probabilitites, and equal costs of misclassification, then the likelihood-ratio test yields the optimal decision boundaries (those which minimize the expected risk). In the two-class case, the resulting decision rule is, if

$$\frac{f(x|\omega_1)}{f(x|\omega_2)} \geq 1, \qquad x \in \omega_1$$
$$\frac{f(x|\omega_1)}{f(x|\omega_2)} < 1, \qquad x \in \omega_2$$

where $f(x|\omega_1)$ and $f(x|\omega_2)$ are the probability density functions (pdf's) for the two classes. The decision boundary of (2-2) can then be expressed as

$$f(x|\omega_1) - f(x|\omega_2) = 0$$

But the same result can be obtained if the locations of the maxima and minima of (2-1) can be found.

The algorithm for finding those extrema makes use of the empirical cdf's for the two classes, and assumes that they are both continuous and monotonically increasing. The empirical cdf is defined as

$$F_n(x) = \frac{\text{number of } \{x_1, x_2, \ldots, x_n\} < x}{n}$$

In [9] the asymptotic optimality of the algorithm is proved, that is, that the boundaries obtained from it converge to the optimal ones which would be determined from (2-2). In addition, expressions for the probability of misclassifications are found.

The method described above has the advantage of requiring little *a priori* information: it assumes that the underlying distributions are continuous, which in most cases is not a serious restriction. It is able to deal with multimodal distributions and mulitple decision boundaries.

Two principal disadvantages, however, are noted when an attempt is made to use the method in multidimensional and/or multiclass problems, as would typify the present work. We observe that, for the multidimensional-feature-vector case, the multivariate analog of a function composed as the difference of empirical cdf's cannot be stored in a computer in a form amenable to convenient extrema determination. In the n-class case, the only possibility for implementation of the algorithm would seem to be the committee solution technique [11], which requires all two-class comparisons to be made; then the class with a majority of favorable decisions is selected. For large problems this approach might not be acceptable, however, since $n(n-1)/2$ individual two-class classifiers would be required.

A method is proposed in [9,10] which is an attempt at avoiding the difficulties listed above. Instead of approximating the difference of distribution functions, the procedure partitions the combined rank ordering of the multiclass samples. It is suitable for both multidimensional and multiclass classification. The next section presents the general structure of the classifier, and Section 2.8.2.2 gives the method and an algorithm for determining the specifications of that structure.

2.8.2.1.2 The General Classifier Structure

The basic building block of the classifier is called the basic selection unit (BSU), and as many BSU's as required are interconnected to form the classifier. A typical BSU is shown in Figure 2-1. The vector F is a 1-dimensional feature vector to be classified by the BSU as coming from one of the m+1 pattern classes $\omega_0, \omega_1, \ldots, \omega_m$, where ω_0 denotes indecision. As will be explained in Section 2.8.2.2, if a decisoin is made that the pattern class is a member of $\{\omega_1, \ldots, \omega_m\}$, the procedure stops; if the BSU is unable to make a classification from that set, its decision is ω_0, and the procedure continues. The BSU allows for these two possibilities by having two kinds of output lines: a decision line and a set of selection lines. If ω_0 is the decision made by a particular BSU, then one of q selection lines, $s_i (i=1,\ldots,q)$, from that BSU will become active. A logical one (1) shall be used to indicate an active state of a selection line, i.e., $s_i = 1$: and a logical zero (0) for an inactive state ($s_i=0$). We denote by $s_{i,j,k}$ the ith selection line from the jth BSU in the kth layer of the classifier (see Figure 2-2). The selection line performs the function of triggering the active state (1) of a BSU in the next (i.e., (k+1)st) layer of the classifier. Unless the selection line entering a BSU is active (s=1), that BSU will remain inactive, and no processing will occur within it.

The decision line $d_{j,k}$ originates at the jth BSU in the kth layer, and is active only if the BSU has decided that the pattern represented by F is an element of $\{\omega_1, \ldots, \omega_m\}$. Thus, the allowable states for $d_{j,k}$ are

$$d_{j,k} = \begin{cases} i, \text{ if class } \omega_i \text{ is decided, } i=1,\ldots,m \text{ (active states)} \\ 0, \text{ if class } \omega_0 \text{ is decided (inactive state).} \end{cases}$$

The BSU, then, serves to either (1) reach a decision as to which of m classes $\omega_1, \ldots, \omega_m$ should be assigned to the input, or (2) decide which subsequent BSU should repeat the process. For any given input to a structure of BSU's there can be at most one BSU which has an active decision line. Other constraints on BSU operation also follow from the structure definitions given above, and can be summarized by the following expressions.

(1) $$s_{i,j,k} = \begin{cases} 0 & \text{inactive state,} \\ 1 & \text{active state,} \end{cases} \qquad \text{all } i,j, \text{ and } k.$$

(2) $$\sum_{i=1}^{q} s_{i,j,k} \leq 1, \quad \text{all } j \text{ and } k.$$

This says that, at most, one selction line per BSU can be active.

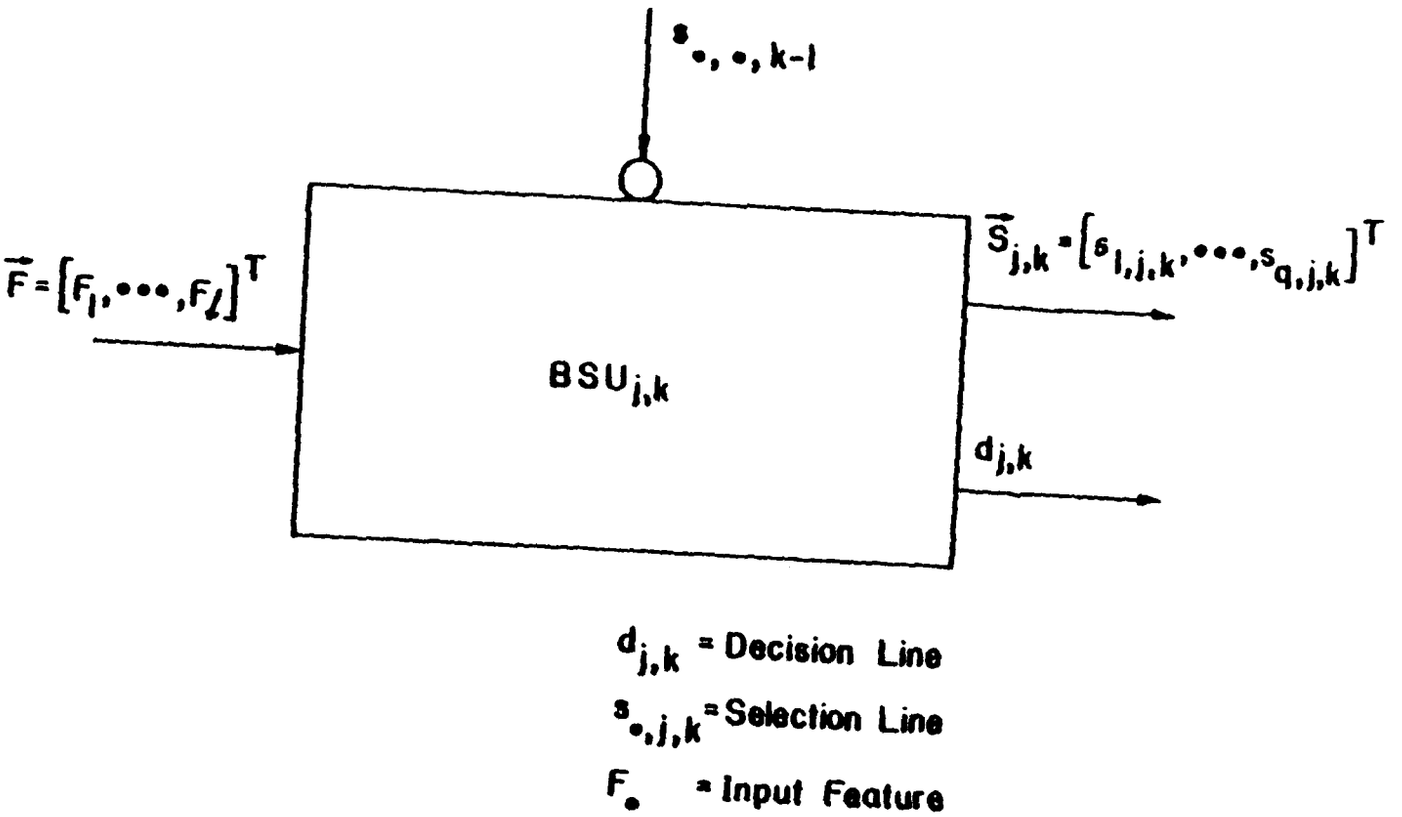


FIGURE 2-1. THE BASIC SELECTION UNIT

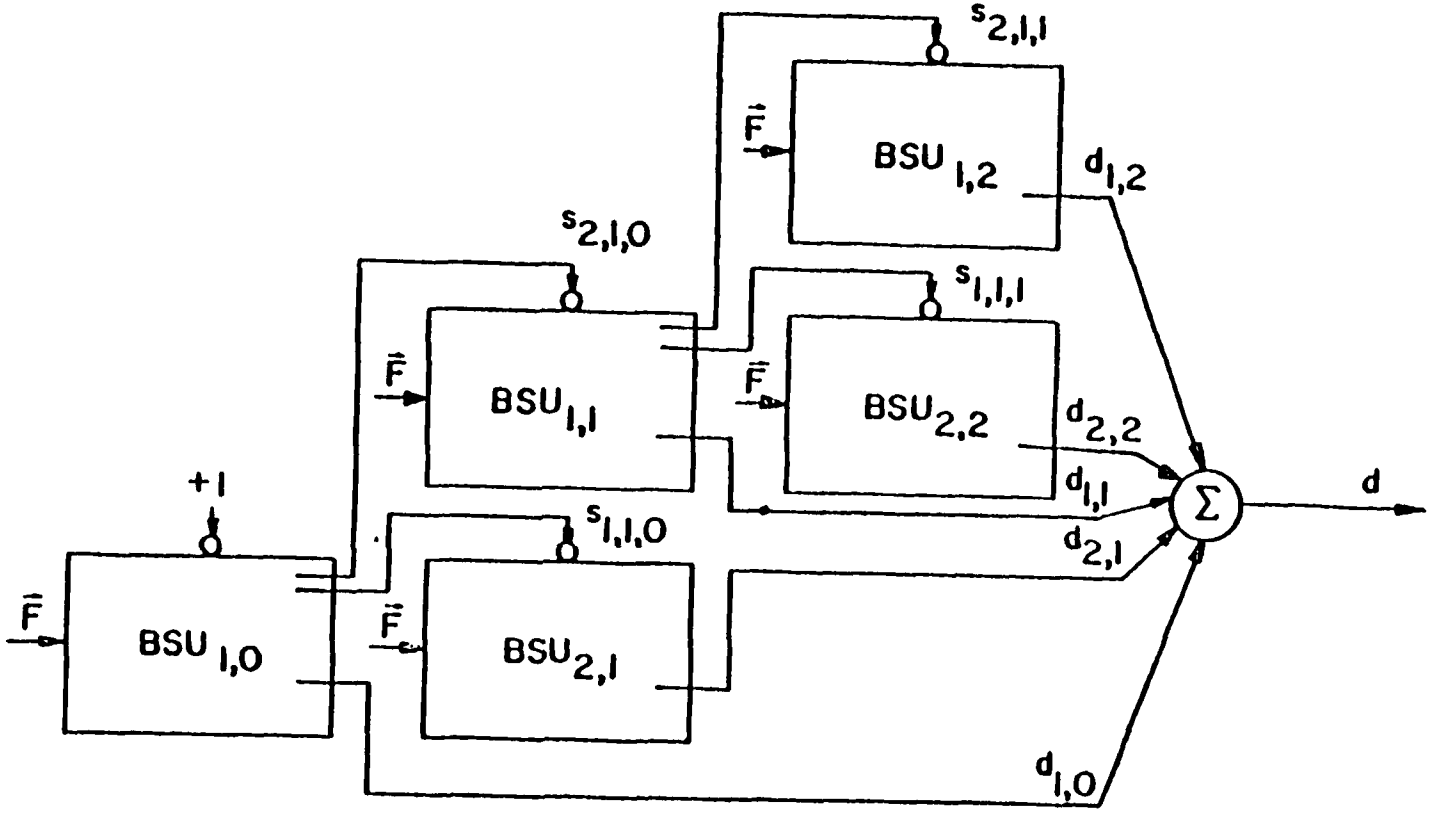


FIGURE 2-2. STRUCTURE OF A TYPICAL CLASSIFIER

$$(3) \quad \left(\sum_{i=1}^{q} s_{i,j,k} \right) \cdot d_{j,k} = 0$$

The decision line and selection line states are mutually exclusive.

$$(4) \quad \left(\sum_{i=1}^{q} s_{i,j,k} + d_{j,k} \right) > 0 \quad \text{if } S_{\ldots,k-1}=1$$

A BSU is inhibited unless there is an active selection line input to it from the previous layer.

Figure 2-3 indicates the conceptual structure of a BSU. Each of its three main components as well as the interconnection scheme is discussed in the next section.

2.8.2.2 Construction of the Classifier

2.8.2.2.1 Introduction

The three subunits of a BSU, as shown in Figure 2-3, are the transgeneration box (optional), which augments the incoming feature F by forming new features which are combinations of the components of F; the component selection box selects one of the features (original or transgenerated) as an input x to the threshold unit. The threshold unit activates one of the q selection lines, or the decision line, depending on the value of x.

In the following discussions the decision lines, selection line labels, and component selection box will be omitted from the structure diagrams. A circled number next to a terminal region in the threshold unit shall indicate which pattern class is selected. The algorithm which follows determines which feature component x should be selected in each BSU, and the intervals, corresponding to the domain of x, which should be connected to the selection and decision lines.

2.8.2.2.2 The Algorithm

Let $X=\{x_1,\ldots,x_{n1}\}$ be a set of n_1 independent observations from class ω_1, and let $Y = \{y_1,\ldots,y_{n2}\}$ be a set of n_2 independent observations from class ω_2. Let K and θ be two prespecified parameters determined from the combined sample size $n=n_1+n_2$. (Some guidelines for the choices of K and θ are discussed below.)

Step 1. Order the combined sample set X+Y according to increasing numerical value to form an ordered set Z. Partition the set Z into successive groups of K samples. (See Figures 2-4(a) - 2-4(c).)

Step 2. For each group count the total number of x's and y's and assign a class label as follows:

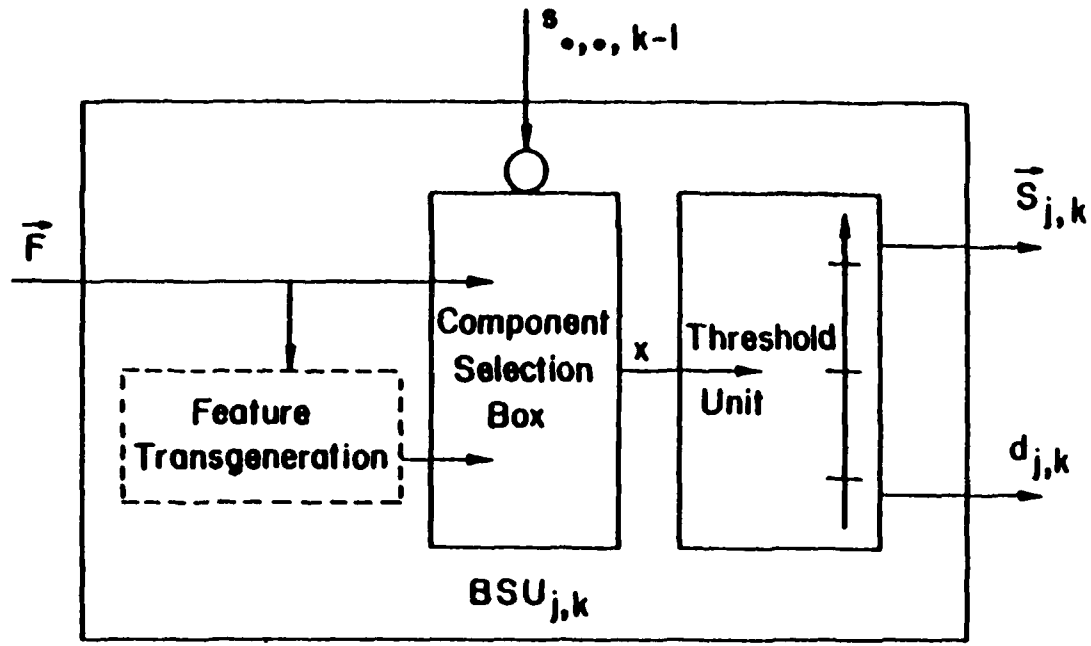


FIGURE 2-3. CONCEPTUAL INTERNAL STRUCTURE OF A BSU

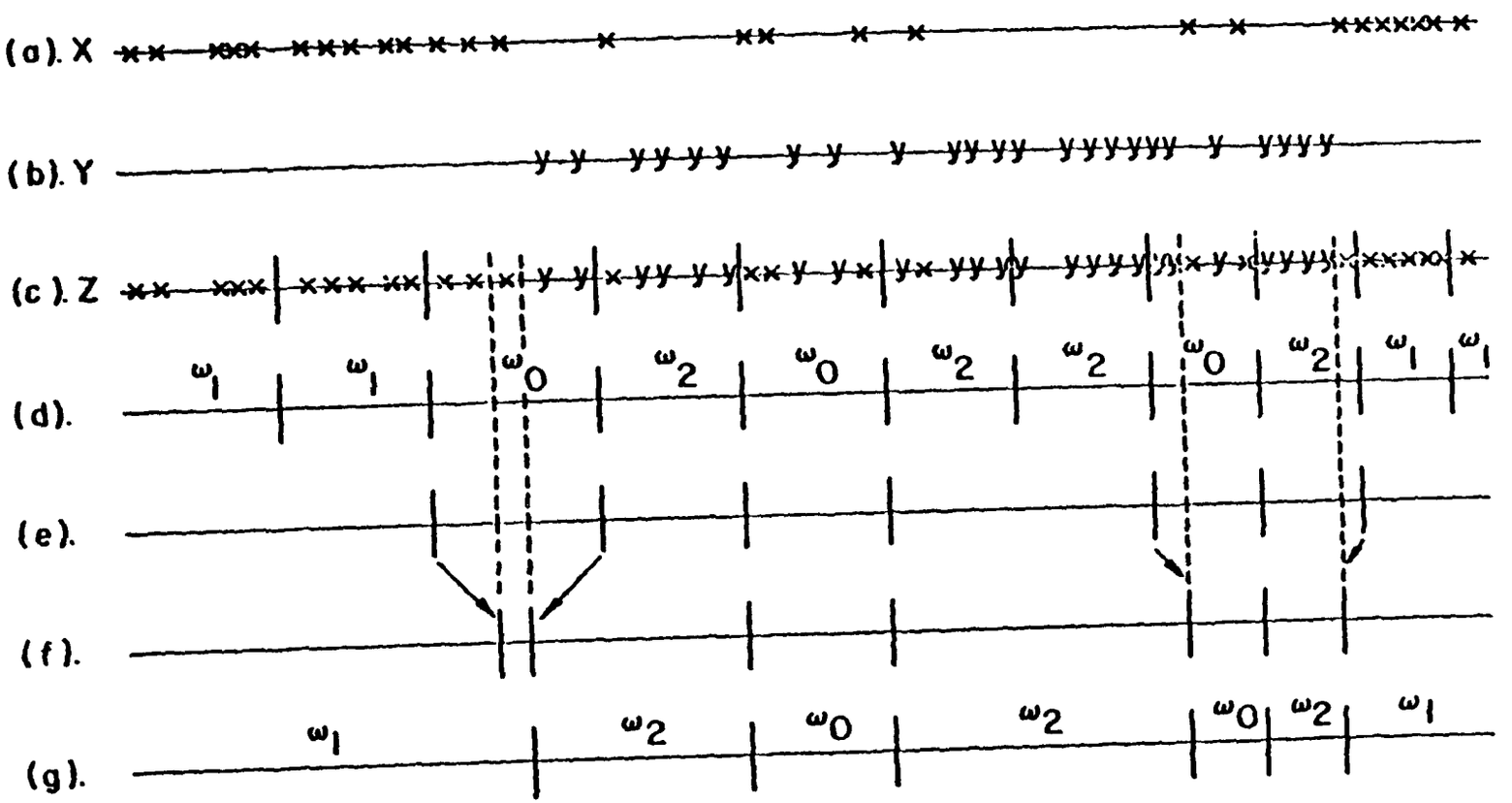


FIGURE 2-4. ILLUSTRATION OF BASIC ALGORITHM WITH SCORE = 41, K = 5, θ = 3.

If number of x's > θ, assign class ω_1
number of y's > θ, assign class ω_2
otherwise, assign class ω_0

Then merge adjacent regions which were assigned the same class (Figures 2-4(d) - 2-4(e)).

Step 3. Adjust the boundaries by perturbing them a maximum of K/2 samples in either direction and relocate them at positions where the most improvement in classification accuracy is obtained (Figure 2-4(f)).

Step 4. If fewer than K/2 samples remain in any one region, dissolve that region and place its samples (preserving the rank order) among the neighboring two regions so as to yield the least increase in misclassification (Figure 2-4(g)).

Step 5. Repeat Step 2. For this final partition compute the empirical classification probability, or SCORE, as

SCORE = number of samples correctly classified.

This training procedure thus yields a set of thresholds to which the (single) feature value of an unknown input sample would be compared: its class would then be assigned according to the label of the region along the "feature axis" in Figure 2-4(g) in which it fell.

The procedure can be extended to the case of a multidimensional feature space by applying the algorithm to each feature separately, and the SCORE (from Step 5) for each component is recorded). The dimension associated with the highest SCORE is selected as the dominant dimension. The observation space is then partitioned by parallel hyperplanes which have the dominant dimension as the common normal vector and intersect it at the boundaries determined by the algorithm for this feature. For each region formed by the above partitions, the procedure is repeated until no new regions are produced. Each time one or more regions of class ω_0 (indecision) in a layer of the classifier are subdivided, an additional layer is added, and this is what produces the selection lines of Figure 2-2.

As mentioned in Section 2.8.2.1.1, one of the disadvantages of the extrema-determination algorithm was that the extension to the multiclass case would involve a substantial amount of additional calculation.

That disadvantage does not exist for the algorithm just discussed: the extension is direct. The original procedure for labeling the partitioned regions was: if

number of x's > θ, assign ω_1
number of y's > θ, assign ω_2
otherwise, assign ω_0.

This step can be reformulated by considering n pattern classes $\omega_1,\ldots,\omega_n$ (letting ω_0 continue to denote indecision). Let $x_{\omega i}$ represent an observation whose true class is ω_i. Then the reformulated assignment rule for each block is: if

number of $x_{\omega i}$'s > θ for some i, assign ω_i
otherwise, assign ω_0

The classifier structure of Figure 2-2 remains as it was in the two-class case.

2.8.2.2.3 Comments on the use of the Algorithm

The choice of K in Step 1 of the algorithm is not of concern in the concept of the procedure, but does play a part in the implementation of the method. The purpose of partitioning the combined sample set (of size n, say) is to reduce the number of partitions which need be considered. This also has the effect of placing an upper bound on the complexity of the classifier at the outset, since the first BSU in the classifier will not require more than K thresholds, and subsequent BSU's (if any) will, in general, have fewer thresholds than the first. The only constraint required on K, therefore, is that it does not increase as fast as n does, i.e.,

$$\lim_{n\to\infty} \frac{k}{n} = 0$$

In the experiments described below, K was chosen proportional to $N^{1/2}$.

An obvious condition on the value of θ used in Step 2 is that $\theta \geq K/2$. If $\theta < K/2$, then conceivably two or more classes would satisfy the inequality governing the assignment of classes to the groups of samples, resulting in an ambiguous procedure. Henrichon [9] has empirically found that good results are obtained with $\theta = 0.6K + (\Delta)$, where $0 \leq \Delta \leq 5$. This approximation, with various values of Δ, was used in the experiments presented below.

The SCORE computed in Step 5 of the algorithm is used in the multidimensional case to determine the order in which features should be chosen by the component selection boxes (Figure 2-3) of the BSU's. As specified in Step 5, the SCORE was simply the number of correctly classified samples. This is a reasonable approach as long as the number of training samples per class remains the same from feature to feature. In some situations, however, the number of samples available per feature varies. A straightforward solution in that case is to use a normalized SCORE, i.e., a percentage. Where necessary, then, the SCORE will be defined as

$$\text{SCORE} = 100 \times \frac{\text{number of correctly classified samples}}{\text{total number of samples}}$$

2.8.3 Examples

To test the design process for the two kinds of classifiers described above, we must use data from some real cases. Because the segmentation routine is still under development, we have chosen to use some data that describe ultrasound signals acquired during examination of human livers. The goal is to use features extracted from the signals to classify a liver as normal or abnormal (in this case, hepatitis).

Four features were used:

(1) d, the average spacing between scatterers
(2) γ, the ratio of the specular to the diffuse backscatter intensities
(3) σ_S, the ratio of the standard deviation of the specular backscatter to the diffuse backscatter
(4) α, the attenuation coefficient.

In the following, the four features are abbreviated, respectively, as D,R, V, and A.
The classifier designs were the following:

(1) Parametric case
The Bayes classifier was used under two alternative cases: (a) that the covariance matrices were different for the different classes (resulting in a nonlinear boundary in feature space), and (b) that the matrices were the same (resulting in a linear boundary). The design data were separated by class for estimation of the covariance matrices in the first case, and pooled in the second case.

(2) Nonparametric Case
The design algorithm was followed, for various values of K and θ.

The results for all of these trials are presented in the next section.

2.9 Classifier Testing

The probability of error is the key quantity in pattern recognition: the estimation of that quantity, therefore, deserves special consideration. There are two kinds of problems. The first is the estimation of the probability of error from available samples, assuming that a classifier is given. The second is the estimation of the probability of error for given distributions. For this problem, the probability of error depends on the classifier to be used as well as on the distributions. Therefore, we first have to specify the nature of the classifier (e.g., the Bayes classifier for minimum error that was defined above): the task then becomes one of finding a way to use available samples for designing the classifier and evaluating the error. Since we have only a finite number of samples, we cannot design the optimum classifier, and the parameters of the classifer are, therefore, also random variables. Furthermore, based on this random classifer, we have to estimate the probability of error.
If we assume that sufficient data are available for estimating accurately the class-conditional density (and distribution) functions of our features, then classical methods exist [12] for estimating the probability of error from N samples drawn from those distributions. Those methods will be employed, with random sampling, to estimate error probabilities and their confidence intervals for the Army data.

To continue the examples presented in Sec. 2.8, however, we must use the second of our approaches, namely, that for the limited-data case. When N samples are given without a classifier design, we have to use those samples to design a classifier as well as to test it. The probability of error to be estimated depends on the given distributions and on the classifier to be used. A number of useful theoretical results have been obtained [13] for the case of the Bayes classifier for minimum error: we will use those results not only for the Bayes classifiers but also for the nonparametric classifiers, all as described in Sec. 2.8.

We described two approaches: (1) N samples are used to design the Bayes classifier and the same N samples are tested. This method has been shown [13] to yield an optimistic bias of the probability of error; (2) N samples are used to design the Bayes classifier, and the samples from the true distributions are used for testing. This method also yields a biased estimate of the probability of error, but the bias is such that the expected value is an upper bound. The samples from the true distribution, however, may be replaced by the samples which are not used to design the classifier and which are independent of them. As the number of test samples increases, the distributions of the test samples tend toward the true distributions.

There are several ways to realize this second approach. The first is to divide available samples into two groups and use one of them for designing the classifier and the other for testing. The second is a refinement of the first; we take out one sample, design a classifier by using N-1 samples, and test the unused sample. This is called the leaving-one-out method. This operation is repeated N times and the number of misclassified samples is counted. The proportion of the total that that number represents is then the estimate of the probability of error. A disadvantage of this method is that N classifiers must be designed. For the nonparametric case, however, it is the only effective conservative approach.

Appendix B contains confusion matrices for the examples of Sec. 2.8, computed using both methods -- testing using the design set, and leaving-one-out.

3. Results

We now consider the results of applying these classifiers to the two kinds of liver disease. Three different kinds of classifiers are used for each of the four features. In addition, all possible subsets of the four features are used to evaluate the performance of multiple measurements. In all cases, a standard format is used: the confusion matrix. The confusion matrix has as its rows the names of the two correct classes and as its columns the names of the classes of the decisions made by the classifier. Hence, an ideal classifier would have all entries on the diagonal, indicating perfect classification. We compute error probability as the total number of misclassified samples divided by the total number of samples.

The three classifier cases we consider are: the leave-one-out case with the Bayes rule for normally distributed data with equal covariance matrices. (Note that the fourth equality of the covariance matrices by pooling the data for the covariance calculation.):

the second case keeps the two classes separate and computes covariance matrices individually. Again, the terms "linear" and "nonlinear" are used because of the nature of the decision boundary that results from the two kinds of covariance matrices. The third kind of classifier again uses two separately-computed covariance matrices but now the design set of data is also used as the testing set. As was noted earlier, this yields an optimistic estimate of error probability for the classifier.

Appendix B presents the three confusion matrices for each feature and combination of features. A total of 47 samples was used: eighteen from the normal class and twenty-nine from the abnormal. Note the relatively high probabilities of error for all three classifiers for certain features and groups of features. These relatively high values are due in part to the error associated with estimating parameters of the probability density function of a small number of samples. We may contrast these results with the second set shown in Appendix B; here we have 79 samples total. Note that the errors are in general smaller.

In any pattern recognition problem, feature selection is a very important step. From a set of candidate features -- even after they have been de-correlated -- one generally seeks to use the smallest number sufficient to achieve the desired error probability. Several recent results [14,15] make it clear that in order to choose the best subset of any given size of an initial candidate set of features, one must examine all possible subsets. In fact, it is possible to do arbitrarily badly if the search is non-exhaustive. In light of those results, we examined the fifteen possible subsets of our four features. Figures 1 and 2 in Appendix B illustrate the variation in probability of error for different kinds of classifiers when different subsets of features are used. For example, in the case in which we have 47 samples all together, we note that the two best features using the nonlinear round-robin or leave-one-out classifier were R and D, but the best set of two features were D and V. This illustrates the fact that in general, the best two features are not necessarily the two best. Note also that if we instead use the linear leave-one-out classifier shown in the left of each set of three bars in the figure, that the two best features are D and V, and that the best are also D and V. Thus we may generalize to say from this example that the best two are not necessarily the two best. That is true as well in the case of three-at-a-time, where the best three are A, R and D, but the three best are D, R, and V. It therefore is important to perform an exhaustive search of all possible subsets if the goal is to find the best performance at a given number of features. In the 79-sample case, in both of the leave-one-out cases (the left bar and center bar of each triplet) we see that the two best features are V and D and that the best two features are also D and V. Note also that the best triplets, D,R,V and D,V,A, are both composed of the best pair, D,V. In the previous data set (the 47-sample case) the best triplet, D,R,V as measured by the nonlinear leave-one-out method was indeed composed of the best two (D,V); but when we examine the linear leave-one-out case, there is a tie for the best set (D,V,A and D,R,A), the latter of which is composed neither of the best two nor of the second best two.

We are led to conclude that any kind of step-wise search for a best subset of features will not in general produce optimal results. This experimental conclusion supports the theoretical results cited earlier.

The non-parametric procedure was applied to the set of 79 samples. The confusion matrices that resulted are shown in Appendix B. Notice that for the features used individually, i.e., stopping the process at the first stage, yields results not far different from what we achieved earlier. In the case, however, where we had two features being used, that is, either D and V or D and R, our performance improved considerably (0.14 error probability). Although the method has not yet been tested with the leave-one-out procedure, that programming is nearly complete and is expected to yield similar results.

4. Conclusions

Work to date has yielded a set of tools that will now work well once features have been extracted from the actual time-series data that we have. The experimental results on the liver ultrasound data are very encouraging and we believe that there will be good performance once the physiological data are analyzed. This set of tools, along with the segmentation routines, should yield results early in the third year of this study.

5. Plans for the Coming Year

During the coming year we expect to segment all of the time-series data, independently for both the activity and the heart-rate data. Separately, we will use a hierarchical approach, referred to earlier in this report, in which we will allow the segments apparent in the activity data to guide the segmentation of the heart-rate data. Again, the rationale for this is that the activity data are in general cleaner than the heart-rate data. Once a set of segments has been established the features that are extracted will be submitted to the classifiers that have been described above. The classification accuracy, concordance with physiological truth (as supplied by the Army), the number of features, and their ease of extraction all will be evaluated with the goal of eventually constructing a very simple signal-processing system for determination of physiological state.

6. Bibliography

[1] M.N. Levy, P. Martin, "Parasympathetic Control of the Heart," Nervous Control of Cardiovascular Function, Oxford University Press, 1984.

[2] R.D. Wurster, "Central Nervous System Regulation of the Heart: An Overview", Nervous Control of Cardiovascular Function, Oxford University Press, 1984.

[3] C.H. Chen, "On a Segmentation Algorithm for Seismic Signal Analysis," Geoexploration, Vol. 23, 35-40.

[4] L. Marple, "A New Autoregressive Spectrum Analysis Algorithm," IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-28, No. 4, 441-454, 1980.

[5] U. Appel, A.V. Brandt, "Adaptive Sequential Segmentation of Piecewise Stationary Time Series," Information Sciences, Vol. 29, 27-56, 1983.

[6] R.O. Duda, P.E. Hart, *Pattern Classification and Scene Analysis*, John Wiley and Sons, New York, 1973.

[7] M.R. Anderberg, *Cluster Analysis for Applications*, Academic Press, New York, 1973.

[8] J.A. Hartigan, *Clustering Algorithms*, John Wiley and Sons, New York, 1975.

[9] E.G. Henrichon, Jr., "On Nonparametric Methods for Pattern Recognition," Ph.D. Thesis, Purdue University, Lafayette, Ind., January 1969.

[10] M.H. Loew and K.S. Fu, "Computer-Aided Medical Diagnosis Using Sequential Pattern-Recognition Techniques," Purdue University, School of Electrical Engineering, Rept. No. TR-EE-72-14, May 1972.

[11] C.M. Ablow and D.J. Kaylor, "A Committee Solution of the Pattern Recognition Problem," IEEE Trans. Information Theory, Vol. IT-11, No. 3, July 1965, pp. 453-455.

[12] W.H. Highleyman, "The design and analysis of pattern recognition experiments," Bell Sys. Tech. Journal, Vol. 41, 1962, pp. 723 - 744.

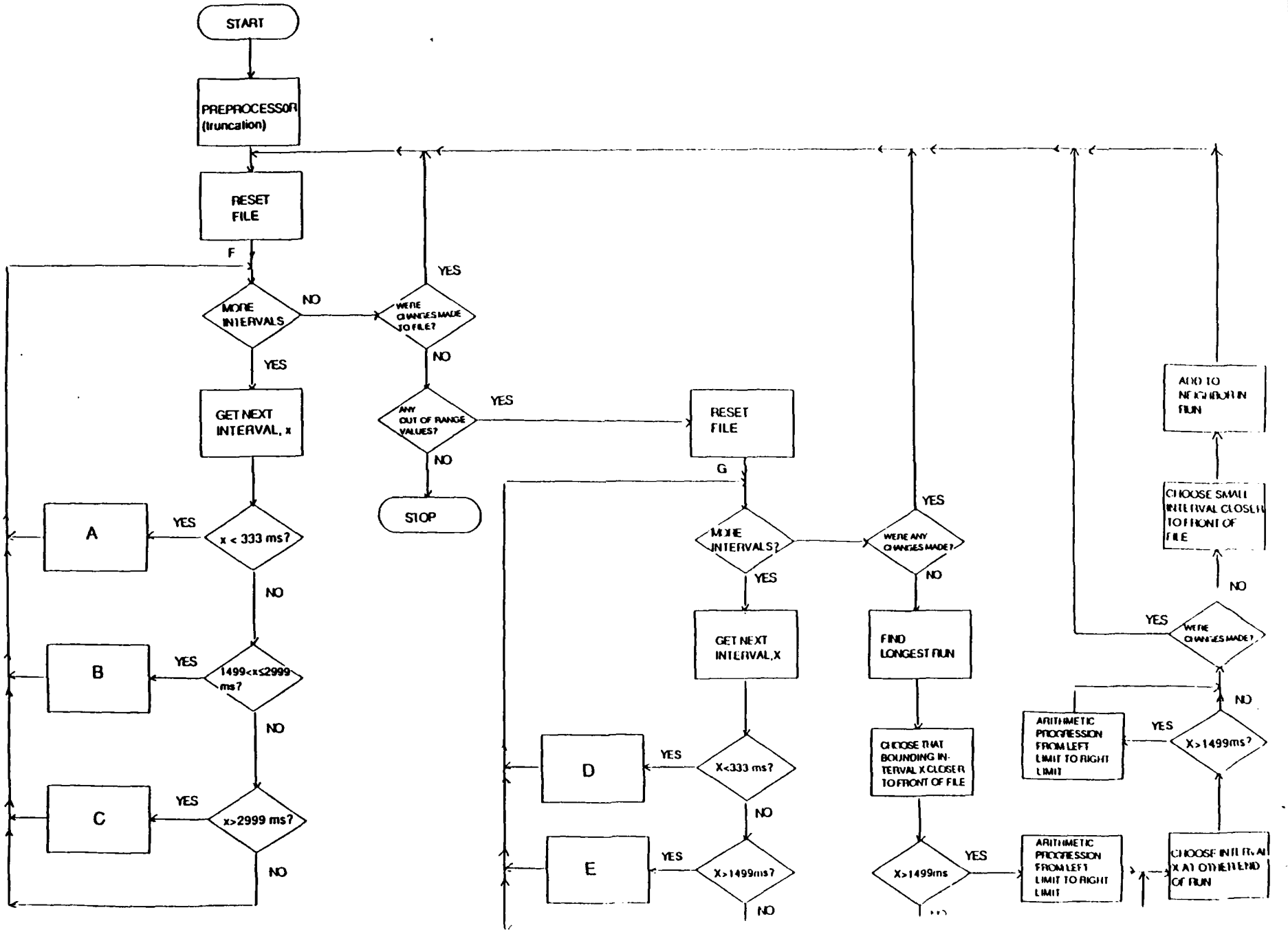
[13] K. Fukunaga, *Introduction to Statistical Pattern Recognition*, Academic Press, New York, 1972.

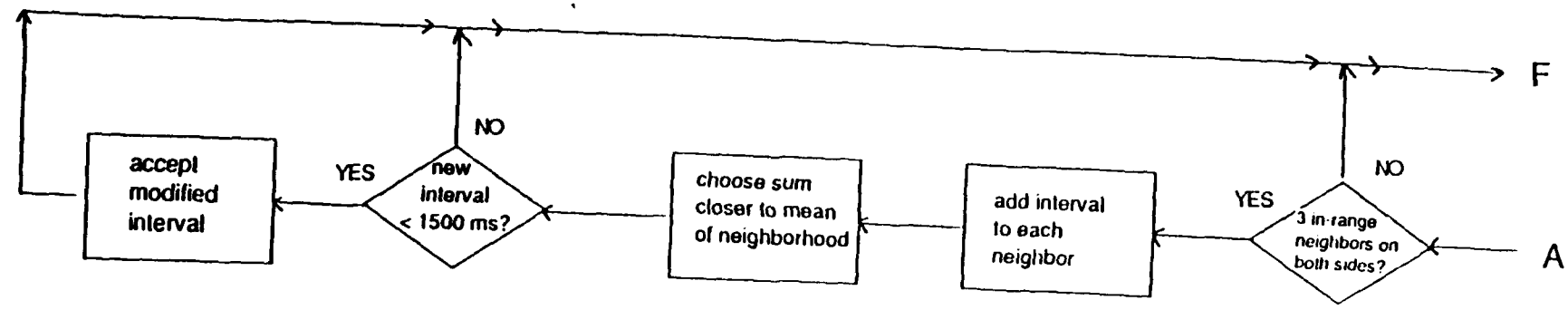
[14] T.M. Cover, and J.M. Van Campenhout, "On the possible orderings in the measurement selection problem", IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-7, No. 9, September 1977, pp. 657-661.

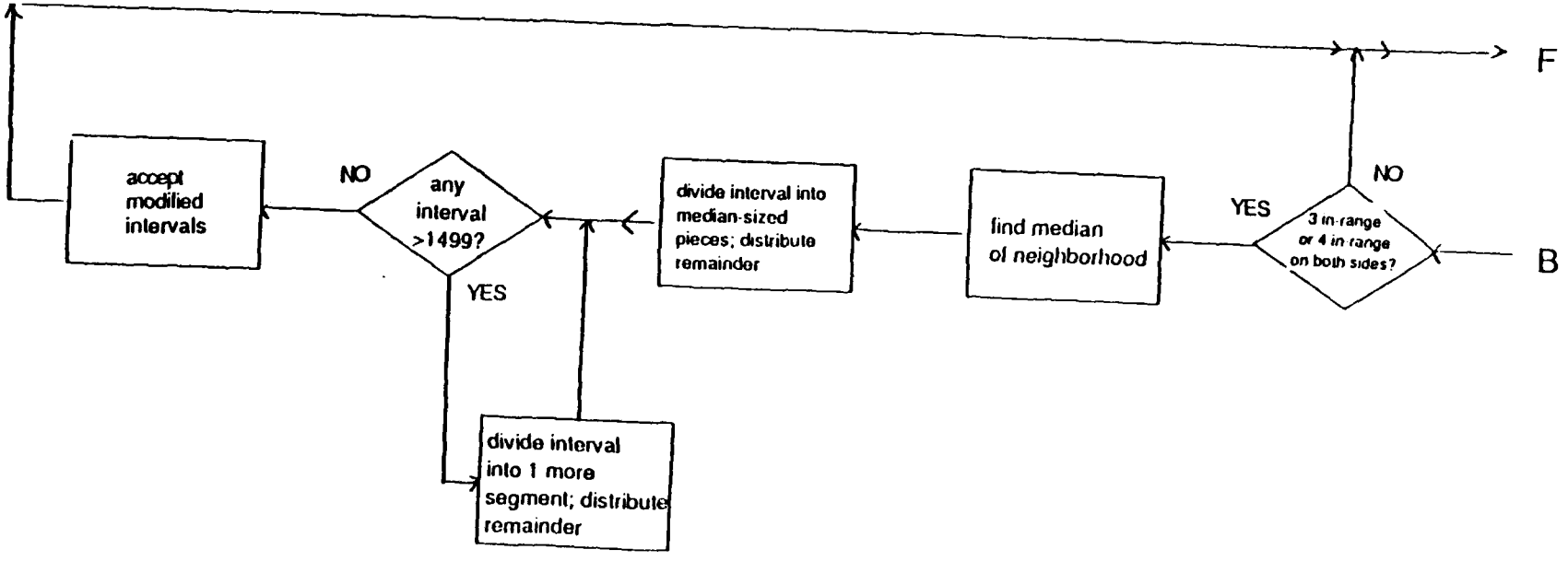
[15] J.M. Van Campenhout, "The arbitrary relation between probability of error and measurement subset," Journal of the American Statistical Association, Vol. 75, No. 369, March 1980, pp. 104-109.

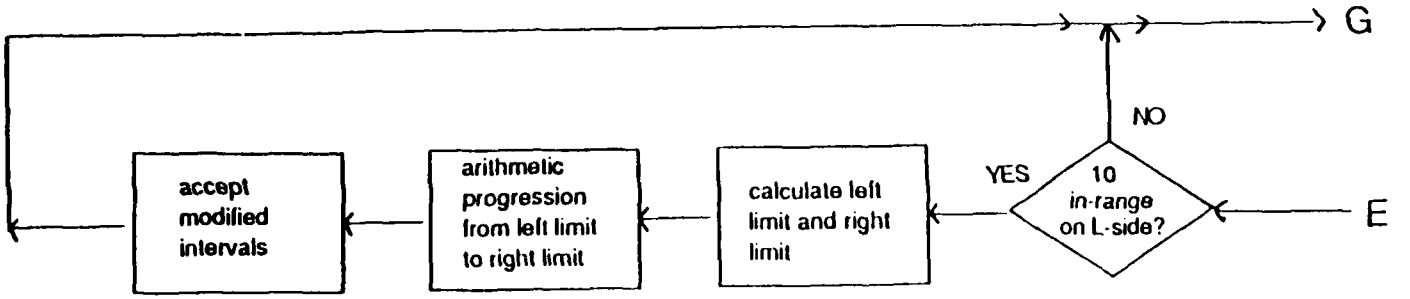
APPENDIX A

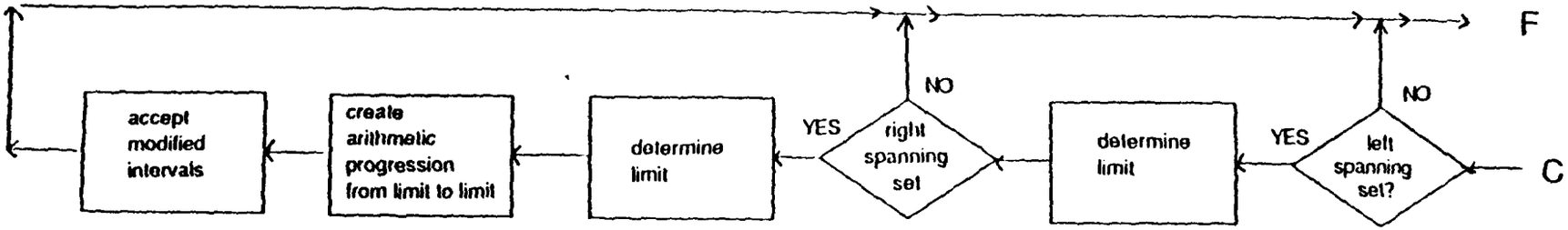
Flowcharts for preprocessing and noise removal program

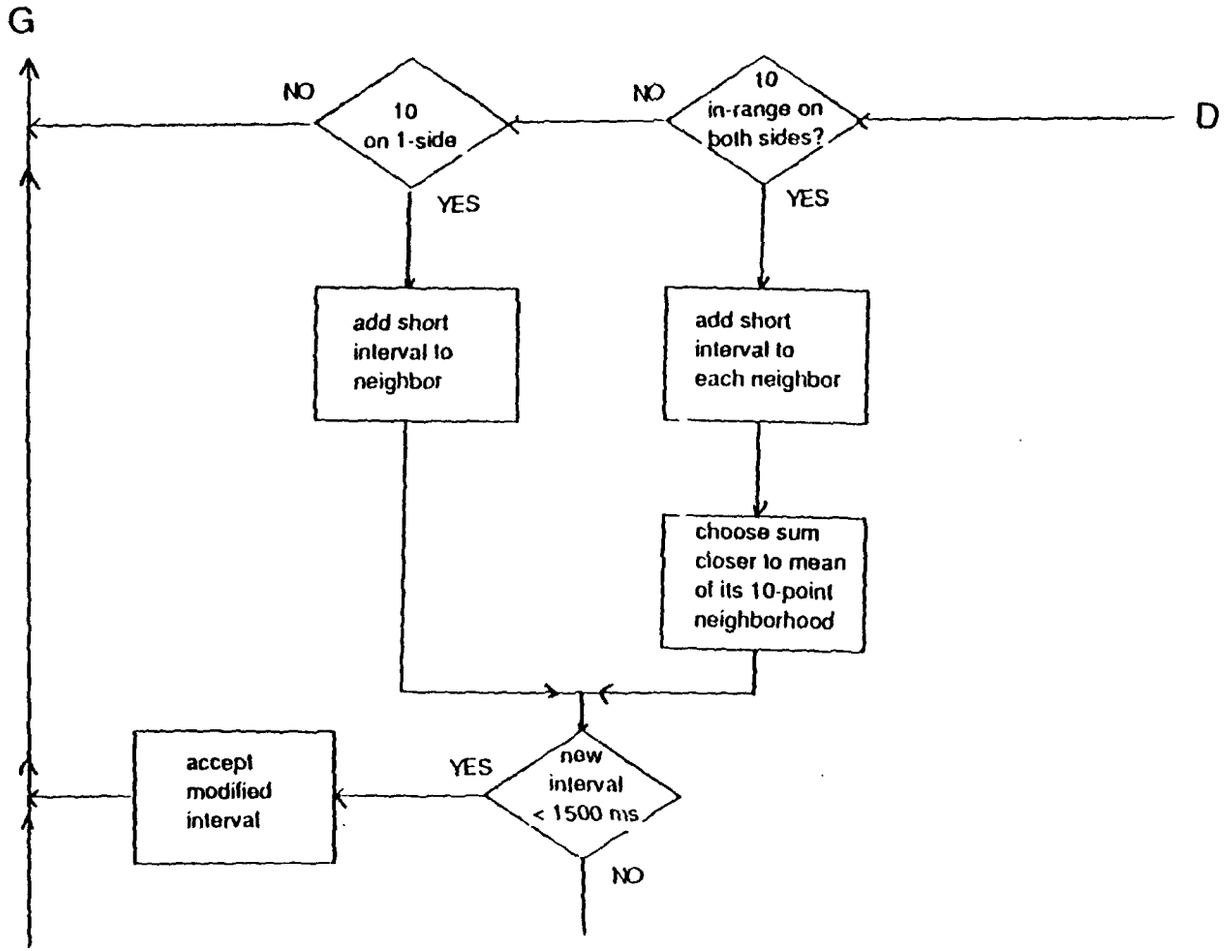

START
PREPROCESSOR (truncation)
RESET FILE
F
MORE INTERVALS
NO
YES
WERE CHANGES MADE TO FILE?
YES
NO
GET NEXT INTERVAL, x
ANY OUT OF RANGE VALUES?
YES
NO
STOP
RESET FILE
G
A
YES
x < 333 ms?
NO
B
YES
1499<x≤2999 ms?
NO
C
YES
x>2999 ms?
NO
MORE INTERVALS?
YES
GET NEXT INTERVAL,X
D
YES
X<333 ms?
NO
E
YES
X>1499ms?
NO
WERE ANY CHANGES MADE?
YES
NO
FIND LONGEST RUN
CHOOSE THAT BOUNDING INTERVAL X CLOSER TO FRONT OF FILE
X>1499ms
YES
ARITHMETIC PROGRESSION FROM LEFT LIMIT TO RIGHT LIMIT
CHOOSE INTERVAL X AT OTHER END OF RUN
ARITHMETIC PROGRESSION FROM LEFT LIMIT TO RIGHT LIMIT
YES
X>1499ms?
NO
WERE CHANGES MADE?
YES
NO
CHOOSE SMALL INTERVAL CLOSER TO FRONT OF FILE
ADD TO NEIGHBORING RUN


F
NO
accept modified interval
YES
new interval < 1500 ms?
choose sum closer to mean of neighborhood
add interval to each neighbor
NO
YES
3 in-range neighbors on both sides?
A


F
accept modified intervals
NO
any interval >1499?
YES
divide interval into median-sized pieces; distribute remainder
find median of neighborhood
NO
YES
3 in-range or 4 in-range on both sides?
B
divide interval into 1 more segment; distribute remainder


G
NO
YES
10
in-range
on L-side?
E
calculate left
limit and right
limit
arithmetic
progression
from left limit
to right limit
accept
modified
intervals


F
NO
NO
YES
YES
C
accept modified intervals
create arithmetic progression from limit to limit
determine limit
right spanning set
determine limit
left spanning set?


G
NO
NO
D
10 on 1-side
10 in-range on both sides?
YES
YES
add short interval to neighbor
add short interval to each neighbor
choose sum closer to mean of its 10-point neighborhood
accept modified interval
YES
new interval < 1500 ms
NO

APPENDIX B

Classifier Testing Results

CLASSIFIERS RESULTS FOR FEATURE(S): V

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	10	8	18
ABNORMAL	7	22	29

ERROR = $\frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$

ERROR = $\frac{15}{47} = .32$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	11	7	18
ABNORMAL	17	12	29

ERROR = $\frac{24}{47} = .51$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	13	5	18
ABNORMAL	14	15	29

ERROR = $\frac{19}{47} = .40$

CLASSIFIERS RESULTS FOR FEATURE(S): **Attenuation**

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	12	6	18
ABNORMAL	15	14	29

$$\text{ERROR} = \frac{21}{47} = .45$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	12	6	18
ABNORMAL	15	14	29

$$\text{ERROR} = \frac{21}{47} = .45$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	12	6	18
ABNORMAL	15	14	29

$$\text{ERROR} = \frac{21}{47} = .45$$

CLASSIFIERS RESULTS FOR FEATURE(S): R

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	10	8	18
ABNORMAL	13	16	29

$$\text{ERROR} = \frac{27}{47} = .45$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	8	10	18
ABNORMAL	10	19	29

$$\text{ERROR} = \frac{20}{47} = .43$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	10	8	18
ABNORMAL	9	20	29

$$\text{ERROR} = \frac{17}{47} = .36$$

CLASSIFIERS RESULTS FOR FEATURE(S): D

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	8	21	29

$$\text{ERROR} = \frac{13}{47} = .28$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	8	21	29

$$\text{ERROR} = \frac{13}{47} = .28$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	8	21	29

$$\text{ERROR} = \frac{13}{47} = .28$$

CLASSIFIERS RESULTS FOR FEATURE(S): V-A

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	12	6	18
ABNORMAL	16	13	29

$$\text{ERROR} = \frac{22}{47} = .47$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	10	8	18
ABNORMAL	14	15	29

$$\text{ERROR} = \frac{22}{47} = .47$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	10	8	18
ABNORMAL	12	17	29

$$\text{ERROR} = \frac{20}{47} = .43$$

CLASSIFIERS RESULTS FOR FEATURE(S): **RV**

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	9	9	18
ABNORMAL	10	19	29

$$\text{ERROR} = \frac{19}{47} = .40$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	12	6	18
ABNORMAL	12	17	29

$$\text{ERROR} = \frac{18}{47} = .38$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	14	4	18
ABNORMAL	8	21	29

$$\text{ERROR} = \frac{12}{47} = .26$$

CLASSIFIERS RESULTS FOR FEATURE(S): R-A

$$\text{ERROR} = \frac{\text{\# of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	11	7	18
ABNORMAL	11	18	29

$$\text{ERROR} = \frac{18}{47} = .38$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	10	8	18
ABNORMAL	11	18	29

$$\text{ERROR} = \frac{19}{47} = .40$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	10	8	18
ABNORMAL	9	20	29

$$\text{ERROR} = \frac{17}{47} = .36$$

CLASSIFIERS RESULTS FOR FEATURE(S): **DR**

$$\text{ERROR} = \frac{\text{\# of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	11	18	29

$$\text{ERROR} = \frac{16}{47} = .34$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	11	18	29

$$\text{ERROR} = \frac{16}{47} = .34$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	14	4	18
ABNORMAL	11	18	29

$$\text{ERROR} = \frac{15}{47} = .32$$

CLASSIFIERS RESULTS FOR FEATURE(S): D-A

ERROR = $\frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	9	20	29

$ERROR = \frac{14}{47} = .30$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	12	6	18
ABNORMAL	10	19	29

$ERROR = \frac{16}{47} = .34$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	9	20	29

$ERROR = \frac{14}{47} = .30$

CLASSIFIERS RESULTS FOR FEATURE(S): D.V

$$\text{ERROR} = \frac{\text{\# of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	8	21	29

$$\text{ERROR} = \frac{13}{47} = .28$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	9	20	29

$$\text{ERROR} = \frac{14}{47} = .30$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	7	22	29

$$\text{ERROR} = \frac{12}{47} = .26$$

CLASSIFIERS RESULTS FOR FEATURE(S): R-V-A

$$\text{ERROR} = \frac{\text{\# of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	11	7	18
ABNORMAL	13	16	29

$$\text{ERROR} = \frac{20}{47} = .43$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	11	7	18
ABNORMAL	11	18	29

$$\text{ERROR} = \frac{18}{47} = .38$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	11	7	18
ABNORMAL	8	21	29

$$\text{ERROR} = \frac{15}{47} = .32$$

CLASSIFIERS RESULTS FOR FEATURE(S): D-R-A

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	9	20	29

$$\text{ERROR} = \frac{14}{47} = .30$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	12	6	18
ABNORMAL	10	19	29

$$\text{ERROR} = \frac{16}{47} = .34$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	9	20	29

$$\text{ERROR} = \frac{14}{47} = .30$$

CLASSIFIERS RESULTS FOR FEATURE(S): D-V-A

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	12	6	18
ABNORMAL	8	21	29

$$\text{ERROR} = \frac{14}{47} = .30$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	12	6	18
ABNORMAL	9	20	29

$$\text{ERROR} = \frac{15}{47} = .32$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	15	3	18
ABNORMAL	7	22	29

$$\text{ERROR} = \frac{10}{47} = .21$$

CLASSIFIERS RESULTS FOR FEATURE(S): **DRV**

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	12	6	18
ABNORMAL	9	20	29

$$\text{ERROR} = \frac{15}{47} = .32$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	7	22	29

$$\text{ERROR} = \frac{12}{47} = .26$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	6	23	29

$$\text{ERROR} = \frac{11}{47} = .23$$

CLASSIFIERS RESULTS FOR FEATURE(S): D-R-V-A

$$\text{ERROR} = \frac{\text{\# of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	13	5	18
ABNORMAL	8	21	29

$$\text{ERROR} = \frac{14}{47} = .30$$

NON-LINEAR ROUND-ROBIN

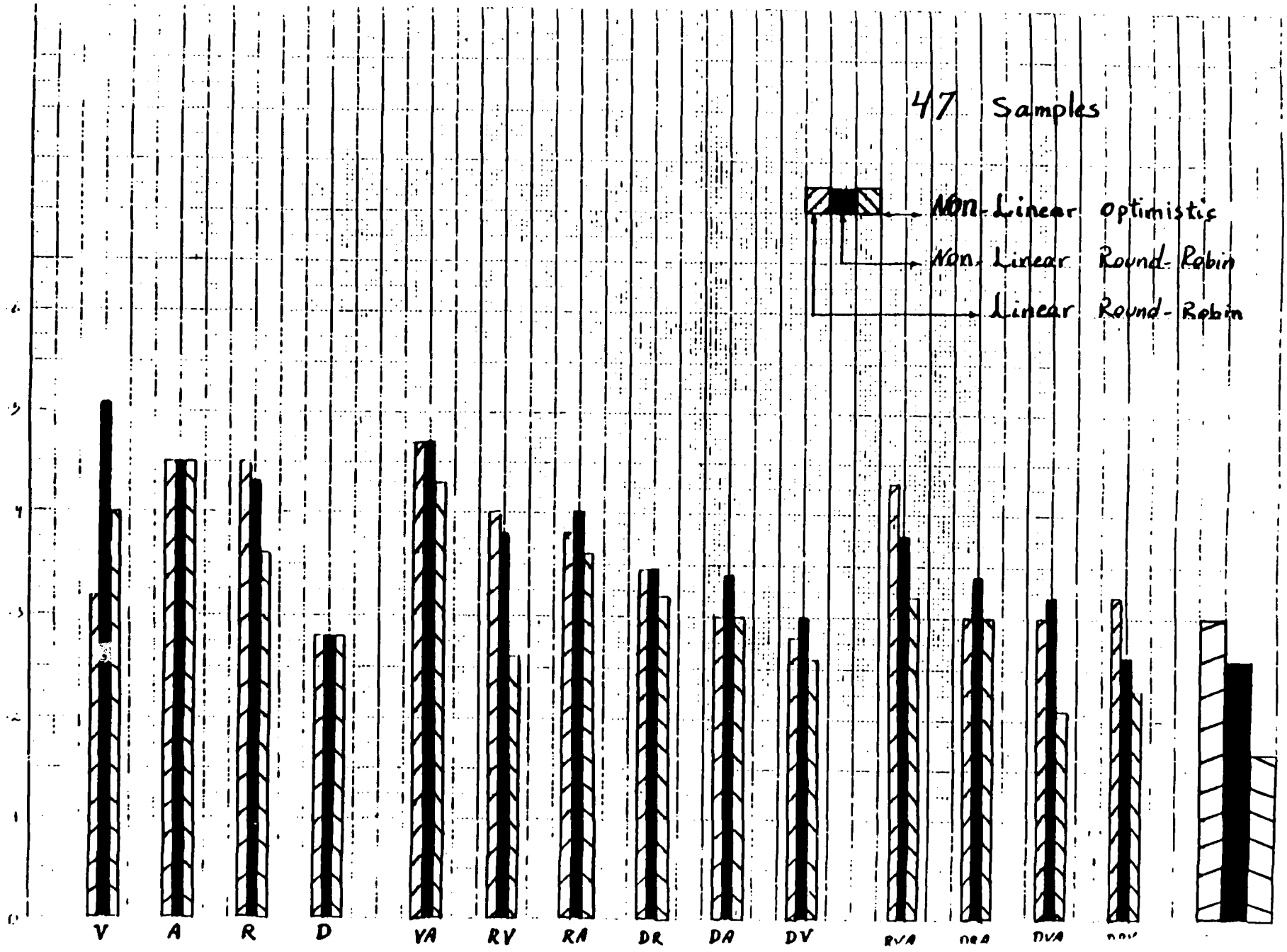
TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	12	6	18
ABNORMAL	6	23	29

$$\text{ERROR} = \frac{12}{47} = .26$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	14	4	18
ABNORMAL	4	25	29

$$\text{ERROR} = \frac{8}{47} = .17$$


47 Samples
Non-Linear Optimistic
Non-Linear Round-Robin
Linear Round-Robin
V
A
R
D
VA
RV
RA
DR
DA
DV
RVA
DVA

CLASSIFIERS RESULTS FOR FEATURE(S): $V_o \; \theta_s \; \frac{\theta_s}{I_d}$

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

True Class \ Decision Class	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	20	11	31
ABNORMAL	9	39	48

$$\text{ERROR} = \frac{20}{79} = .25$$

NON-LINEAR ROUND-ROBIN

True Class \ Decision Class	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	20	11	31
ABNORMAL	11	37	48

$$\text{ERROR} = \frac{22}{79} = .28$$

OPTIMISTIC NON-LINEAR

True Class \ Decision Class	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	20	11	31
ABNORMAL	11	37	48

$$\text{ERROR} = \frac{22}{79} = .28$$

CLASSIFIERS RESULTS FOR FEATURE(S): A

ERROR = $\frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	17	14	31
ABNORMAL	19	29	48

$ERROR = \frac{33}{79} = .42$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	17	14	31
ABNORMAL	19	29	48

$ERROR = \frac{33}{79} = .42$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	17	14	31
ABNORMAL	19	29	48

$ERROR = \frac{33}{79} = .42$

CLASSIFIERS RESULTS FOR FEATURE(S): $R = \frac{\bar{I}_s}{I_d}$

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	20	11	31
ABNORMAL	17	31	48

$$\text{ERROR} = \frac{28}{79} = .35$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	22	9	31
ABNORMAL	22	26	48

$$\text{ERROR} = \frac{31}{79} = .39$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	22	9	31
ABNORMAL	22	26	48

$$\text{ERROR} = \frac{31}{79} = .39$$

CLASSIFIERS RESULTS FOR FEATURE(S): D

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	25	6	31
ABNORMAL	9	39	48

$$\text{ERROR} = \frac{15}{79} = .19$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	25	6	31
ABNORMAL	13	35	48

$$\text{ERROR} = \frac{19}{79} = .24$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	27	4	31
ABNORMAL	13	35	48

$$\text{ERROR} = \frac{17}{79} = .22$$

CLASSIFIERS RESULTS FOR FEATURE(S): VA

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	23	8	31
ABNORMAL	11	37	48

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

$$\text{ERROR} = \frac{19}{79} = .24$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	23	8	31
ABNORMAL	11	37	48

$$\text{ERROR} = \frac{19}{79} = .24$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	23	8	31
ABNORMAL	11	37	48

$$\text{ERROR} = \frac{19}{79} = .24$$

CLASSIFIERS RESULTS FOR FEATURE(S): R-V

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	23	8	31
ABNORMAL	10	38	48

$$\text{ERROR} = \frac{18}{79} = .23$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	25	6	31
ABNORMAL	15	33	48

$$\text{ERROR} = \frac{21}{79} = .27$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	26	5	31
ABNORMAL	14	34	48

$$\text{ERROR} = \frac{19}{79} = .24$$

CLASSIFIERS RESULTS FOR FEATURE(S): RA

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	18	13	31
ABNORMAL	16	32	48

$$\text{ERROR} = \frac{29}{79} = .37$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	17	14	31
ABNORMAL	18	30	48

$$\text{ERROR} = \frac{32}{79} = .41$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	18	13	31
ABNORMAL	16	32	48

$$\text{ERROR} = \frac{29}{79} = .37$$

CLASSIFIERS RESULTS FOR FEATURE(S): D-R

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	26	5	31
ABNORMAL	11	37	48

$$\text{ERROR} = \frac{16}{79} = .20$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	27	4	31
ABNORMAL	13	35	48

$$\text{ERROR} = \frac{17}{79} = .22$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	27	4	31
ABNORMAL	13	35	48

$$\text{ERROR} = \frac{17}{79} = .22$$

CLASSIFIERS RESULTS FOR FEATURE(S): DA

ERROR = $\frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	26	5	31
ABNORMAL	10	38	48

ERROR = $\frac{15}{79} = .19$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	25	6	31
ABNORMAL	11	37	48

ERROR = $\frac{17}{79} = .22$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	28	3	31
ABNORMAL	10	38	48

ERROR = $\frac{13}{79} = .16$

CLASSIFIERS RESULTS FOR FEATURE(S): D-V

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	27	4	31
ABNORMAL	9	39	48

$$\text{ERROR} = \frac{13}{79} = .16$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	27	4	31
ABNORMAL	10	38	48

$$\text{ERROR} = \frac{14}{79} = .18$$

PTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	27	4	31
ABNORMAL	9	39	48

$$\text{ERROR} = \frac{13}{79} = .16$$

CLASSIFIERS RESULTS FOR FEATURE(S): RVA

ERROR = # of misclassified / Total # of samples

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	22	9	31
ABNORMAL	11	37	48

$ERROR = \frac{20}{79} = .25$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	24	7	31
ABNORMAL	12	36	48

$ERROR = \frac{19}{79} = .24$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	24	7	31
ABNORMAL	9	39	48

$ERROR = \frac{16}{79} = .20$

CLASSIFIERS RESULTS FOR FEATURE(S): DRA

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	25	6	31
ABNORMAL	11	37	48

$$\text{ERROR} = \frac{17}{79} = .22$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	25	6	31
ABNORMAL	11	37	48

$$\text{ERROR} = \frac{17}{79} = .22$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	26	5	31
ABNORMAL	9	39	48

$$\text{ERROR} = \frac{14}{79} = .18$$

CLASIFIERS RESULTS FOR FEATURE(S): DVA

$$\text{ERROR} = \frac{\text{\# of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	27	4	31
ABNORMAL	10	38	48

$$\text{ERROR} = \frac{14}{79} = .18$$

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	26	5	31
ABNORMAL	10	38	48

$$\text{ERROR} = \frac{15}{79} = .19$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	27	4	31
ABNORMAL	10	38	48

$$\text{ERROR} = \frac{14}{79} = .18$$

CLASSIFIERS RESULTS FOR FEATURE(S): D-R-V

ERROR = $\frac{\text{\# of misclassified}}{\text{Total \# of samples}}$

Linear Round-Robin

TRUE CLASS \ DECISION	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	27	4	31
ABNORMAL	9	39	48

ERROR = $\frac{13}{79}$ = .16

NON-LINEAR ROUND-ROBIN

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	27	4	31
ABNORMAL	11	37	48

ERROR = $\frac{15}{79}$ = .19

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	27	4	31
ABNORMAL	9	39	48

ERROR = $\frac{13}{79}$ = .16

CLASSIFIERS RESULTS FOR FEATURE(S): DRVA

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

Linear Round-Robin

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	26	5	31
ABNORMAL	10	38	48

$$\text{ERROR} = \frac{15}{79} = .19$$

NON-LINEAR ROUND-ROBIN

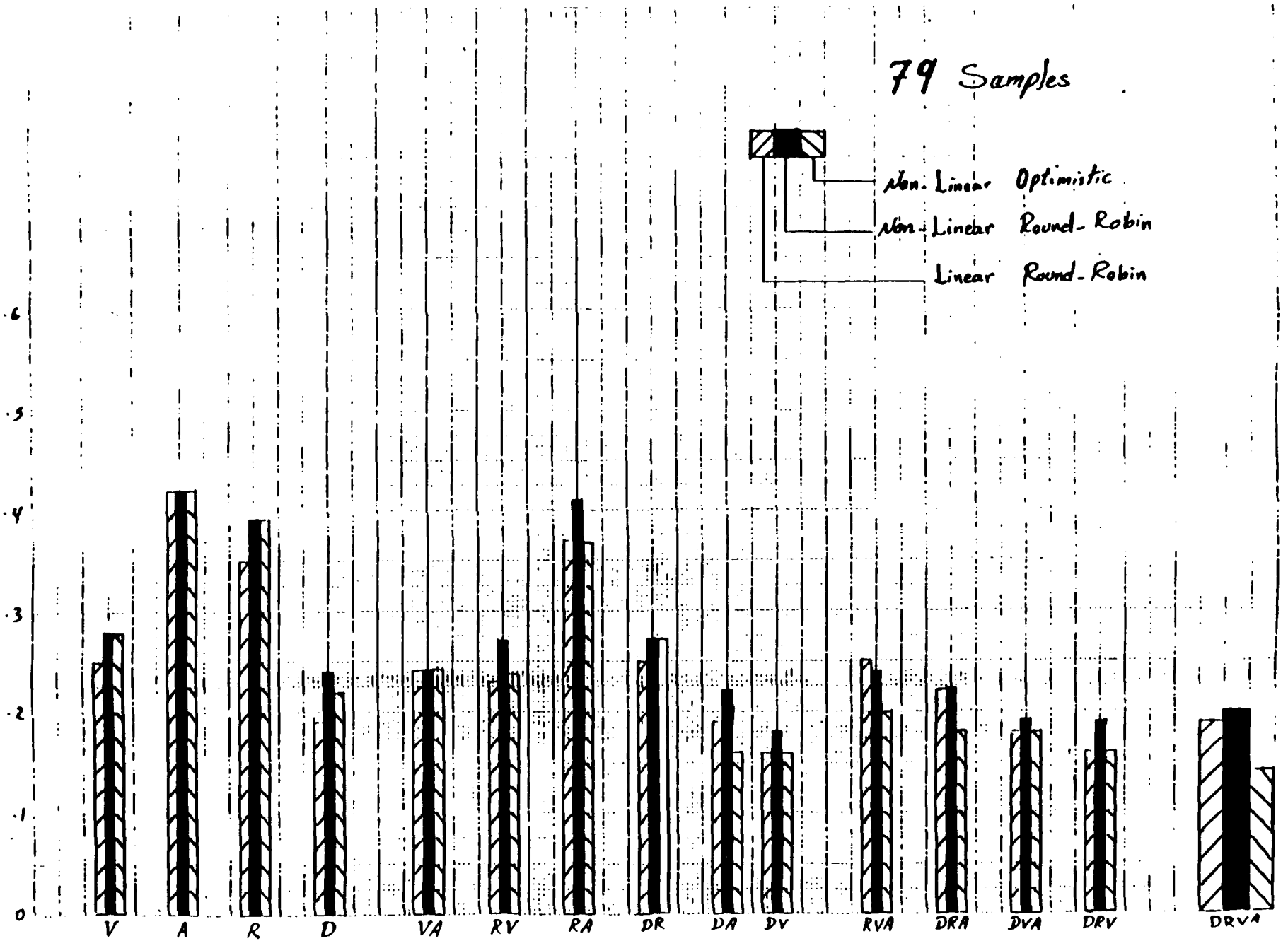
TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	24	7	31
ABNORMAL	9	39	48

$$\text{ERROR} = \frac{16}{79} = .20$$

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN CLASS
NORMAL	26	5	31
ABNORMAL	6	42	48

$$\text{ERROR} = \frac{11}{79} = .14$$


79 Samples
Non-Linear Optimistic
Non-Linear Round-Robin
Linear Round-Robin
.6
.5
.4
.3
.2
.1
0
V
A
R
D
VA
RV
RA
DR
DA
DV
RVA
DRA
DVA
DRV
DRVA

CLASSIFIERS RESULTS FOR FEATURE(S) in: Non-parametric

$$\text{ERROR} = \frac{\#\text{ of misclassified}}{\text{Total \# of samples}}$$

A

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	24	7	31
ABNORMAL	9	39	48

$$\text{ERROR} = \frac{16}{79} = .20$$

V

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	19	12	31
ABNORMAL	6	42	48

$$\text{ERROR} = \frac{18}{79} = .23$$

R

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	15	16	31
ABNORMAL	6	42	48

$$\text{ERROR} = \frac{22}{79} = .28$$

CLASSIFIERS RESULTS FOR FEATURE(S) in Non-Parametric

$\overline{d}/r$

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	21	10	31
ABNORMAL	1	47	48

$\overline{d}/r_{pool}$

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL	22	9	31
ABNORMAL	2	46	48

OPTIMISTIC NON-LINEAR

TRUE CLASS \ DECISION CLASS	NORMAL	ABNORMAL	TOTAL NO. OF SAMPLES IN TRUE CLASS
NORMAL			
ABNORMAL			

ERROR = $\frac{\text{\# of misclassified}}{\text{Total \# of samples}}$

ERROR = $\frac{11}{79} = .14$

ERROR = $\frac{11}{79} = .14$

ERROR =

APPENDIX C

Preprocessing and Noise Removal Program

```
PROGRAM EKGCleanOutput;

(* This is an experimental program developed to clean noise points out of
   EKG data files.  The program produces hard-copy of various statistics,
   and a new disk file containing the cleaned data.

   Investigators    Murray H. Loew
                    Linda Sibert

   Programmer       Lee Spector

   Supporting Institution    The George Washington University, Washington D.C.

   Project Funding    U.S. Army, Dr. Redmond ...

                                                                          *)

CONST  MaxSweep = 100;

TYPE  DataArray = Array[1..10000] of Integer;
      NeighArray = Array[1..9] of Integer;

      SpanArray = Array[1..100] of Integer;  { New structure for Bubble Sort }
                                             { Added : 7/28/86              }

VAR  SweepNumber, Datum, SmallBad, MediumBad, LargeBad, NumBad, NumFinal,
           Index, NumPumpPrimes, DirtyNum, CleanNum, ScratchNum : Integer;
     NoChanges   Boolean;
     DirtySet, CleanSet, ScratchSet : DataArray;
     SFail1Neigh, SFail3Neigh, (* SFailDev DEACTIVATED 3/14/86, *) SFailLge,
     SmallFixed, MFailNeigh, LFailNeigh, LFailVar
                                          : Array[1..MaxSweep] of Integer;
     MFixed4, MFixed3 : Array[1..MaxSweep, 15..29] of Integer;
     LFixed   Array[1..MaxSweep, 6..32] of Integer;




Procedure InitStats;

(* This procedure initializes the variables which hold statistical
   information regarding the cleaning process.                            *)

VAR Index, Index2 : Integer;

Begin
     SmallBad  = 0;
     MediumBad  = 0;
     LargeBad  = 0;
     For Index  = 1 to MaxSweep DO
          Begin
               SFail1Neigh[Index]  = 0;
               SFail3Neigh[Index]  = 0;
               (* SFailDev[Index]  = 0;  DEACTIVATED 3/14/86 *)
               SFailLge[Index]  = 0;
```

```
                SFailFixed[Index] := 0;
                MFailNeigh[Index] := 0;
                LFailNeigh[Index] := 0;
                LFailVar[Index]   := 0;
                For Index2  := 15 to 29 DO
                     Begin
                          MFixed4[Index, Index2] := 0;
                          MFixed3[Index, Index2] := 0
                     End; (* For *)
                For Index2 := 6 to 32 DO
                     LFixed[Index, Index2] := 0
           End (* For *)
End. (* InitStats *)
```

```
Procedure PrintStats;

(* This prucedure prints histograms and other statistical output *)

VAR  Index, I2   Integer;

Begin
Writeln (' ***************************************************************');
Writeln (' XXXXXXXXXXXXXXX    FILE CLEANING STATISTICS    XXXXXXXXXXXXXXX');
Writeln (' ***************************************************************');
Writeln;
Writeln (' ** NOTE **  THESE STATISTICS ARE FOR THE SWEEPS OF THE FILE');
Writeln ('                AFTER ', NumPumpPrimes, ' PUMP-PRIMES !!');
Writeln;
Writeln (' Initial Bad Points in File :');

Writeln ('                                          < 333 (Small) : ',
                                                                SmallBad);

Writeln ('                                >= 1500 and < 3000 (Medium) : ',
                                                               MediumBad);

Writeln ('                                         >= 3000 (Large) : ',
                                                               LargeBad);

Writeln;                                                  TOTAL : ',
Writeln (' Bad Points After Cleaning : ', NumBad);  Writeln;
                                    SmallBad + MediumBad + LargeBad);
For Index := 1 to SweepNumber DO
Begin
  Writeln (' #######   STATISTICS FOR SWEEP NUMBER ', Index, '   #######');
  Writeln;
  Writeln (' ** NOTE **  ', NumPumpPrimes, ' Pump-Primes have taken place.');
  Writeln;
  Writeln (' Small Failures due to BAD NEIGHBORS DISTANCE 1 : ',
                                                    SFailNeigh[Index]);

  Writeln (' Small Failures due to BAD NEIGHBORHOODS OF 3   : ',
                                                    SFail3Neigh[Index]);

  (* Writeln (' Small Failures due to DEVIATION CRITERION      : ',
                                                    SFailDev[Index]); DEACTIVATED 3/14/86 *)

  Writeln (' Small Failures due to NEW VALUE >= 1500        : ',
```

```
                                                    SFailLge[Index]);
  Writeln (' Small Successes : ', SmallFixed[Index]);  Writeln;
  Writeln (' Medium Failures due to BAD NEIGHBORHOODS : ',MFailNeigh[Index]);
  Writeln;
  Writeln (' Medium Successes with Neighborhoods of 4 (* 2) : ');  Writeln;
  Writeln ('         RANGE                    # SUCCESSES');
  Writeln ('     -----------                  -----------');
  For I2  := 15 to 29 DO
       Writeln('      ',100*I2,' - ',100*(I2+1)-1,MFixed4[Index,I2] :25);
  Writeln;
  Writeln (' Medium Successes with Neighborhoods of 3 (* 2) : ');  Writeln;
  Writeln ('         RANGE                    # SUCCESSES');
  Writeln ('     -----------                  -----------');
  For I2  := 15 to 29 DO
       Writeln('      ',100*I2,' - ',100*(I2+1)-1,MFixed3[Index,I2] :25);
  Writeln;
  Writeln(' Large Failures due to BAD NEIGHBORHOODS      : ',LFailNeigh[Index])
;
  Writeln(' Large Failures due to VARIABILITY CRITERION : ',LFailVar[Index]);
  Writeln;
  Writeln(' Large Successes : ');  Writeln;
  Writeln('         RANGE                    # SUCCESSES');
  Writeln('   ------------------             -----------');
  For I2  := 6 to 31 DO
       Writeln((I2*1000)DIV 2 :10,' - ',(((I2+1)*1000)DIV 2)-1 :8,
                                                    LFixed[Index,I2] :21);
  Writeln('     16000 and up     ', LFixed[Index,32] :21);  Writeln
End (* For *)
End. (* PrintStats *)



Function Good(EKGvalue : Integer) : Boolean;

(* This function returns "true" if its argument satisfies
                    333 <= EKGvalue < 1500                              *)

Begin
     Good  := (EKGvalue >= 333) AND (EKGvalue < 1500)
End. (* Good *)



Function Good3Neigh(Ary : DataArray; Point : Integer; N : Integer) : Boolean;

(* This function returns "true" if the pointed-at value has three good
   neighbors on each side                                               *)
```

```
      If (NOT (Point >= 3) AND ((Point + 3) <= N)) Then   (* Point near *)
           Good3Neigh  = False                            (* beginning or EOF *)
      Else Good3Neigh  = (Good(Ary[Point - 3]))
                         AND (Good(Ary[Point - 2]))
                         AND (Good(Ary[Point - 1]))
                         AND (Good(Ary[Point + 1]))
                         AND (Good(Ary[Point + 2]))
                         AND (Good(Ary[Point + 3]))
End. (* Good3Neigh *)
```

```
Function Good4Neigh(Ary : DataArray; Point : Integer; N : Integer) : Boolean;

(* This function returns "true" if the pointed-at value has four good
   neighbors on each side
                                                                        *)
Begin

     If (NOT (Point >= 4) AND ((Point + 4) <= N)) Then   (* Point near *)
          Good4Neigh  = False                            (* beginning or EOF *)
     Else Good4Neigh  = (Good(Ary[Point - 4]))
                        AND (Good(Ary[Point - 3]))
                        AND (Good(Ary[Point - 2]))
                        AND (Good(Ary[Point - 1]))
                        AND (Good(Ary[Point + 1]))
                        AND (Good(Ary[Point + 2]))
                        AND (Good(Ary[Point + 3]))
                        AND (Good(Ary[Point + 4]))

End. (* Good4Neigh *)
```

```
Function GoodLeft10(Index : Integer; Ary : DataArray) : Boolean;

VAR  Counter   Integer;
     Temp   Boolean;
Begin
     Temp  = True;
     For Counter  = 1 to 10 DO
          If (Index - Counter) < 1 Then Temp := False
          Else If NOT Good(Ary[Index-Counter]) Then Temp := False;
     GoodLeft10  = Temp
End. (* GoodLeft10 *)
```

```
Function GoodRight10(Index   Integer; Ary : DataArray; NumPoints : Integer)
                                                                    : Boolean;
VAR  Counter   Integer;
     Temp   Boolean;
Begin
     Temp  := True;
     For Counter  := 1 to 10 DO
          If (Index + Counter) > NumPoints Then Temp := False
          Else If NOT Good(Ary[Index+Counter]) Then Temp := False;
     GoodRight10  := Temp
End. (* GoodRight10 *)
```

```
(* DEACTIVATED 3/14/86
Function WithinDev(Val : Integer; Ary : DataArray; Point : Integer) : Boolean;
*)
(* This function returns "true" if val is within 20 % of the mean of the
   6 points surrounding the "point"ed-at point in ary.                    *)

(* VAR Deviation, Mean : Integer;

Begin
     Mean  := (Ary[Point-3]+Ary[Point-2]+Ary[Point-1]+Ary[Point+1]+Ary[P
oint+2]+
               Ary[Point+3]) DIV 6;
     Deviation  := Mean DIV 5;
     WithinDev  := (Val <= (Mean+Deviation)) AND (Val >= (Mean-Deviation))
End. *) (* WithinDev *)
```

```
Procedure PrintDev(Value, Mean   Integer);
(* ADDED 3/14/86 *)
Begin
Write( ' The new value deviates from the mean of its 6 neighbors by ',
          (100 * (Value - Mean)) DIV Mean, ' %')
End.
```

```
Function MeanoNeigh(Ary   DataArray; Point : Integer) : Integer;

(* This function returns the mean of the six points surrounding the "point"ed
   at point in Ary

Begin                                                                    *)
     MeanoNeigh  = (Ary[Point-3]+Ary[Point-2]+Ary[Point-1]+Ary[Point+1]+
                    Ary[Point+2]+Ary[Point+3]) DIV 6
End. (* MeanoNeigh *)
```

```
Procedure SmallCase(InArray : DataArray; VAR OutArray : DataArray;
                    InPointer : Integer; VAR OutPointer : Integer;
                    NumPoints : Integer; VAR NoChanges, SkipOne : Boolean);
(* This procedure implements the algorithm for cleaning values  < 333.
   GLOBAL VARIABLES REFERENCED :  SmallBad, SmallFixed, SFailINeigh,
                                  SFail3Neigh, SFailDev, SweepNumber
   MODIFIED 3/3/86

VAR  NewValue, NeighMean : Integer;  Success : Boolean;                  *)
Begin
     Success  = False; SkipOne  = False;
     If SweepNumber = 1 Then SmallBad  = SmallBad+1; (* Keep track of # bad *)
     Writeln(' SmallCase Algorithm Activated...  Value = ', InArray[InPointer])
     If (InPointer = 1) OR (InPointer = NumPoints) Then
          Begin
               Writeln (' Value AT beginning or end of file... ');
               SFailINeigh[SweepNumber] := SFailINeigh[SweepNumber] + 1
          End (* If *)
     Else If (NOT Good(InArray[InPointer-1]))OR(NOT Good(InArray[InPointer+1]))
          Then Begin
               Writeln (' Bad neighbor distance = 1 ... ');
               SFailINeigh[SweepNumber] := SFailINeigh[SweepNumber] + 1
          End (* Else If *)
     Else If (NOT Good3Neigh(InArray, InPointer, NumPoints)) Then
          Begin
               Writeln (' Neighborhood of 3 not adequate...  ');
               SFail3Neigh[SweepNumber] := SFail3Neigh[SweepNumber] + 1
          End (* Else If *)
     Else Begin
(* Temp *)     Writeln(' Neighborhood is  ', InArray[InPointer-3], ' ',
```

```
    (* Temp *)       InArray[InPointer-2], ' ', InArray[InPointer-1], ' ',
    (* Temp *)       InArray[InPointer], ' ', InArray[InPointer+1], ' ',
    (* Temp *)       InArray[InPointer+2], ' ', InArray[InPointer+3]);
    (* Temp *)    Writeln(' LeftSum = ', InArray[InPointer]+InArray[InPointer-1]
);
    (* Temp *)    Writeln(' RightSum = ', InArray[InPointer]+InArray[InPointer+1]),
                  NeighMean  = MeanNeigh(InArray, InPointer);
                  if ABS(InArray[InPointer - 1]+InArray[InPointer]-NeighMean)
                   < ABS(InArray[InPointer + 1]+InArray[InPointer]-NeighMean)
                   Then
                       Begin                          (* Add to Preceeding Value *)
                         NewValue := InArray[InPointer-1] + InArray[InPointer
],
(* If WithinDev(NewValue, InArray, InPointer) DEACTIVATED 3/14/86 *)
                         if (NewValue < 1500) Then
                           Begin          (* NewValue OK - Make Change *)
                             OutArray[Outpointer-1] := NewValue;
                             Success := True;
                             PrintDev(NewValue, NeighMean)
                           End (* If *)
(* Else If NewValue < 1500 Then DEACTIVATED 3/14/86
   Begin
   Write(' Proposed new value not within ');
   Writeln('20 % of mean of 6 neighbors... ');
   SFailDev[SweepNumber] := SFailDev[SweepNumber]+1
   End *) (* Else If *)
                         Else Begin
                              Write(' Proposed new value >= 1500... ');
                              SFailLge[SweepNumber] := SFailLge[SweepNumber]+1
                             End (* Else *)
                    End (* If *)
                Else Begin                          (* Add to Succeeding Value *)
                         NewValue := InArray[InPointer+1] + InArray[InPointer
],
(* If WithinDev(NewValue, InArray, InPointer) Then  DEACTIVATED 3/14/86
   Begin *)(* NewValue OK - Make Change *)
                         if NewValue < 1500 Then
                           Begin
                             OutArray[Outpointer] := NewValue;
                             OutPointer := OutPointer + 1;
                             SkipOne := True;
                             Success := True;
                             PrintDev(NewValue, NeighMean)
                           End (* If *)
                         Else Begin
                              Write(' Proposed new value >= 1500... ');
                              SFailLge[SweepNumber] := SFailLge[SweepNumber]+1
                             End (* Else *)
(* End *) (* If *)
(* Else If NewValue < 1500 Then Begin DEACTIVATED 3/14/86
   Write(' Proposed new value not within ');
   Writeln('20 % of mean of 6 neighbors... ');
   SFailDev[SweepNumber] := SFailDev[SweepNumber]+1
   End*) (* Else *)
                    End (* Else *)
          End, (* Else *)
      If Success Then
           Begin
                SmallFixed[SweepNumber] := SmallFixed[SweepNumber] + 1;
                NoChanges  = False,
```

```
                Writeln
            End (* If *)
        Else Begin
                OutArray[OutPointer]  := InArray[InPointer];
                OutPointer  := OutPointer + 1;
                Writeln(' Algorithm Fails');
                Writeln
            End (* Else *)
    End. (* SmallCase *)
```

```
Procedure Bubble9(VAR  Ary : NeighArray; N : Integer);

(* This procedure sorts the N values in Ary.  N <= 9.

VAR  Index1, Index2, Temp   Integer;                                    *)

Begin
     For Index1  := 1 to (N - 1) DO
          For Index2 := 1 to (N - 1) DO
               If Ary[Index2] > Ary[Index2 + 1] Then
                    Begin
                         Temp := Ary[Index2];
                         Ary[Index2]  := Ary[Index2 + 1];   (* Exchange *)
                         Ary[Index2 + 1] := Temp
                    End (* If *)
End. (* Bubble9 *)
```

```
Function GetMedian44(Ary : DataArray; Point : Integer) : Integer;

(* This function returns the median of the 8 neighbors of
   the pointed-at value in Ary.

VAR Neighborhood : NeighArray;                                          *)
    Index   Integer;

Begin
     For Index  := - 4 to 4 DO
          Neighborhood[Index+5]  := Ary[Point + Index];
     Bubble9(Neighborhood, 9);
     GetMedian44  := Neighborhood[5]
End. (* GetMedian44 *)
```

```
Function GetMedian33(Ary  DataArray; Point : Integer) : Integer;

{ This function returns the median of the 6 neighbors of
  the pointed-at value in Ary.

VAR Neighborhood  NeighArray;                                          }
    Index   Integer;

Begin
     For Index  = - 3 to 3 DO
          Neighborhood[Index+4] := Ary[Point + Index];
     Bubble9(Neighborhood, 7);
     GetMedian33  = Neighborhood[4]
End; { GetMedian33 }
```

```
Function GetMedian5(Ary : DataArray; Point : Integer) : Integer;

{ This function returns the median of the 5 points in Ary BEGINNING with
  the pointed-at value in Ary.

VAR Neighborhood : NeighArray;                                         }
    Index   Integer;

Begin
     For Index  = 1 to 5 DO
          Neighborhood[Index]  = Ary[Point + Index - 1];
     Bubble9(Neighborhood, 5);
     GetMedian5  = Neighborhood[3]
End; { GetMedian5 }
```

```
Procedure MediumCase(InArray : DataArray; VAR OutArray : DataArray;
                     InPointer   Integer; VAR OutPointer : Integer;
                     NumPoints   Integer; VAR NoChanges : Boolean);
```

```
(* This is the procedure which implements the algorithm for cleaning values
   1499 and  = 2999
(* GLOBAL VARIABLES ACCESSED   SweepNumber, MediumBad, MFailNeigh, MFixed4,
                               MFixed3                                      *)

VAR  Median, NumPartit, Remainder, Index : Integer;
     Success    Boolean;
     NewValues    Array[1  10] of Integer;

Begin
     If SweepNumber = 1 Then MediumBad := MediumBad+1;  (* Keep track # bad *)
     Writeln(' MediumCase Algorithm Activated... Value = ', InArray[InPointer])

     If Good4Neigh(InArray, InPointer, NumPoints) Then
          Begin
                                                        (* ck neigh of 4 *)
               MFixed4[SweepNumber, InArray[InPointer] DIV 100] :=
                         MFixed4[SweepNumber, InArray[InPointer] DIV 100] +
1,
               Success := True;
               Median := GetMedian44(InArray, InPointer)
          End (* If *)
     Else If Good3Neigh(InArray, InPointer, NumPoints) Then (* ck neigh of 3 *)
          Begin
               MFixed3[SweepNumber, InArray[InPointer] DIV 100] :=
                         MFixed3[SweepNumber, InArray[InPointer] DIV 100] +
1.
               Success := True;
               Median := GetMedian33(InArray, InPointer)
          End (* Else If *)
     Else Success := False;
     If Success Then
          Begin                                         (* Success - split up value *)
               NumPartit := InArray[InPointer] DIV Median;
               Remainder := InArray[InPointer] MOD Median;
               If Remainder >= (Median DIV 2) Then
                    Begin
                         NumPartit := NumPartit + 1;
                         Median := InArray[InPointer] DIV NumPartit;
                         Remainder := InArray[InPointer] MOD Median
                    End; (* If *)
               For Index := 1 to NumPartit DO           (* NewValues = Median *)
                    NewValues[Index] := Median;
               Index := 1;
               While Remainder > 0 DO                   (* Distribute Remainder *)
                    Begin
                         NewValues[Index] := NewValues[Index] + 1;
                         Index := Index + 1;
                         If Index > NumPartit Then Index := 1;
                         Remainder := Remainder - 1
                    End; (* While *)
               For Index := 1 to NumPartit DO           (* Send out new values *)
                    Begin
                         OutArray[OutPointer] := NewValues[Index];
                         OutPointer := OutPointer + 1
                    End; (* For *)
               NoChanges := False;
               Writeln (' Algorithm Succeeds!');
               Writeln
          End (* If *)
```

```
    Else Begin
              OutArray[OutPointer]  := InArray[InPointer];
              OutPointer   := OutPointer + 1;
              MFailNeigh[SweepNumber]  := MFailNeigh[SweepNumber] + 1;
              Writeln (' Algorithm Fails!');
              Writeln
         End (* Else *)
End;  (* MediumCase *)
```

```
Procedure SpanLeft(InArray : DataArray; InPointer : Integer;
                   VAR  LeftSum, LeftMin, LeftMax, LeftMean : Integer;
                   VAR  LeftSetOK : Boolean ;
                   VAR  LastLeftSpan : Integer ); { <-- Added : 7/28/86 }

(* This procedure forms the left spanning-set.                                    *)

VAR  Index, PointsUsed : Integer;

Begin
     Index  := InPointer;
     LeftSum  := 0;
     LeftMin  := 9999;
     LeftMax  := -9999;
     PointsUsed := 0;
     LeftSetOK  := True;
     REPEAT
          If Index = 1 Then
               LeftSetOK := False
          Else If NOT Good(InArray[Index - 1]) Then
               LeftSetOK := False
          Else
               Begin
                    Index := Index - 1;
                    PointsUsed := PointsUsed + 1;
                    If InArray[Index] < LeftMin Then LeftMin := InArray[Index
);
                    If InArray[Index] > LeftMax Then LeftMax := InArray[Index
);
                    LeftSum := LeftSum + InArray[Index]
               End (* Else *)
     UNTIL (NOT LeftSetOK) OR (LeftSum >= InArray[InPointer]);
     If LeftSetOK Then
          Begin
               LeftMean := LeftSum DIV PointsUsed ;
               LastLeftSpan := Index ;   { <-- Added : 7/28/86 }
          End
     Else
          Writeln (' Left Spanning-Set Fails - Sum at failure = ',LeftSum)
End;  (* SpanLeft *)
```

```
Procedure SpanRight(InArray   DataArray; InPointer, NumPoints   Integer;
                    VAR  RightSum, RightMin, RightMax, RightMean : Integer;
                    VAR  RightSetOK : Boolean ;
                    VAR  LastRightSpan : Integer );  { <-- Added : 7/28/86 }

{ This procedure forms the right spanning-set                                  }

VAR  Index, PointsUsed   Integer;

Begin
     Index  := InPointer;
     RightSum  := 0;
     RightMin  := 9999;
     RightMax  := -9999;
     PointsUsed  := 0;
     RightSetOK  := True;
     REPEAT
          If Index = NumPoints Then
               RightSetOK := False
          Else If NOT Good(InArray[Index + 1]) Then
               RightSetOK := False
          Else
               Begin
                  Index  := Index + 1;
                  PointsUsed := PointsUsed + 1;
                  If InArray[Index] < RightMin Then RightMin := InArray[Index
];
                  If InArray[Index] > RightMax Then RightMax := InArray[Index
];
                  RightSum := RightSum + InArray[Index]
               End (* Else *)
     UNTIL (NOT RightSetOK) OR (RightSum >= InArray[InPointer]);
     If RightSetOK Then
          Begin
               RightMean := RightSum DIV PointsUsed ;
               LastRightSpan := Index ;  { <-- Added : 7/28/86 }
          End
     Else
          Writeln (' Right Spanning-Set Fails - Sum at failure = ',RightSum)
End;  (* SpanRight *)




(-------------------------------------------)
(--                                       --)
(--  Procedure for Bubble Sorting array   --)
```

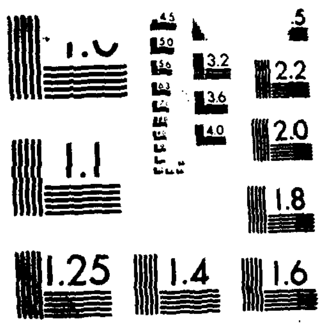
```
{   Added      7/28/86                                  --}
{                                                       --}
{-------------------------------------------------------}
PROCEDURE BubbleSort ( SortArray   SpanArray ;
                       Last        Integer   );

    VAR

       Index1, Index2, Temp   Integer          ;

    BEGIN

        FOR Index1  := 1 to (Last-1) DO
            FOR Index2  := 1 to (Last-1) DO
                IF SortArray[Index2] > SortArray[Index2+1] THEN
                   BEGIN
                       Temp := SortArray[Index2] ;
                       SortArray[Index2] := SortArray[Index2+1] ;
                       SortArray[Index2+1] := Temp ;
                   END                           ;

    END                                          ;
```

```
{-------------------------------------------------}
{--                                             --}
{--  Function for Finding the Right Median      --}
{--  used by the Procedure LargeCase            --}
{--  Added    7/28/86                           --}
{--                                             --}
{-------------------------------------------------}
FUNCTION FindRightMedian (InArray : DataArray ;
                          InPointer : Integer ;
                          LastRightSpan : Integer ) : Integer ;

    VAR

       I, Index, Last, n  Integer                ;
       SortArray   SpanArray                     ;

    BEGIN

       Index  := InPointer                       ;
       Last   := LastRightSpan - InPointer       ;
       FOR I  := 1 to Last DO
           BEGIN
               Index := Index + 1                ;
               SortArray[I] := InArray[Index] ;
           END                                   ;

       BubbleSort( SortArray, Last )             ;
```

AD-A217 670 PATTERN RECOGNITION OF CARDIOVASCULAR AND PSYCHOMOTOR VARIABILITY IN RESP..(U) GEORGE WASHINGTON UNIV WASHINGTON DC DEPT OF ELECTRICAL ENGIN. 2/2

UNCLASSIFIED M H LOEW ET AL. AUG 86 DAMD17-84-C-4129 F/G 6/4 NL

[redacted]


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
3.2
3.6
4.0

```
        IF Odd(Last) THEN
            BEGIN
                n  := Last DIV 2 ;
                FindRightMedian := SortArray[n+1] ;
            END
        ELSE
            BEGIN
                n  := Last DIV 2 ;
                FindRightMedian  := TRUNC((SortArray[n]+SortArray[n+1])/2),
            END (--  IF  --)

    END  (-- FindLeftMedian --)
```

```
{-----------------------------------------------}
{--                                           --}
{--   Function for finding the Left Median    --}
{--   used by the Procedure LargeCase         --}
{--   Added     7/28/86                       --}
{--                                           --}
{-----------------------------------------------}
FUNCTION FindLeftMedian ( InArray : DataArray ;
                          InPointer : Integer ;
                          LastLeftSpan : Integer ) : Integer ;

    VAR

       I, Index, Last, n : Integer ;
       SortArray   SpanArray ;

    BEGIN

       Index  := InPointer ;
       Last   := InPointer - LastLeftSpan ;
       FOR I  := 1 to Last DO
           BEGIN
               Index := Index - 1 ;
               SortArray[I] := InArray[Index] ;
           END ;

       BubbleSort( SortArray, Last ) ;

       IF Odd(Last) THEN
           BEGIN
               n  := Last DIV 2 ;
               FindLeftMedian := SortArray[n+1] ;
           END
       ELSE
           BEGIN
               n  := Last DIV 2 ;
               FindLeftMedian  := TRUNC((SortArray[n]+SortArray[n+1])/2),
           END (--  IF  --) ;
```

```
Procedure LargeCase(InArray : DataArray; VAR OutArray : DataArray;
                    InPointer   Integer; VAR OutPointer : Integer;
                    NumPoints   Integer; VAR NoChanges : Boolean);

(* This procedure implements the algorithm for cleaning values  >= 3000.     *)

VAR  LeftSum, LeftMin, LeftMax, LeftMean, NumPartit,
     RightSum, RightMin, RightMax, RightMean, ComDiff, Index,
     Accum, SmallerMean, StatAryInd, LastLeftSpan, LastRightSpan  : Integer;
     LeftSetOK, RightSetOK, Success : Boolean;
     NewValues : Array[1..100] of Integer;
     RN, AllowedVar, PercVarLeft, PercVarRight : Real;

Begin
     AllowedVar := 0.20;  (* **** SET ALLOWED VARIABILITY HERE **** *)
     Success := False;
     If SweepNumber = 1 Then LargeBad := LargeBad+1;  (* Keep track # bad *)
     Writeln(' LargeCase Algorithm Activated... Value = ', InArray[InPointer]);

     SpanLeft(InArray, InPointer, LeftSum, LeftMin, LeftMax,
              LeftMean, LeftSetOK,LastLeftSpan);

     SpanRight(InArray, InPointer, NumPoints, RightSum, RightMin, RightMax,
               RightMean, RightSetOK,LastRightSpan);

     If LeftSetOK AND RightSetOK Then
          Begin
               PercVarLeft := (1.0*(LeftMax - LeftMin)) / LeftMean;
               PercVarRight := (1.0*(RightMax - RightMin)) / RightMean;

     {--------- This Routine was modified on 7/28/86 ----------}

               IF NOT ( (PercVarLeft >= (0.0 - AllowedVar)) AND
                  (PercVarLeft <= AllowedVar) ) THEN
                  LeftMean := FindLeftMedian(InArray, Inpointer,
                                             LastLeftSpan)   ;


               IF NOT ( (PercVarRight >= (0.0 - AllowedVar)) AND
                  (PercVarRight <= AllowedVar) ) THEN
                  RightMean := FindRightMedian(InArray, Inpointer,
                                             LastRightSpan)  ;

               Success := True  ;

          End (* If *)
     Else
          Begin
               LFailNeigh[SweepNumber] := LFailNeigh[SweepNumber] + 1;
               Writeln(' Spanning-sets not formed... ');
```

```
      print*, 'Calculation of sigma'
      Q = 0.0
      DO 400 J = 1,N
         Q(J) = 0.0
         DO 300 I = 1,N
            print*,A(I,J),CC(I,J)
            Q(J) = Q(J) + E(I)*CC(I,J)
300      CONTINUE
400   CONTINUE
      print*,' '
      DO 500 I = 1,N
         print*,Q(I),E(I)
         S = S + Q(I) * E(I)
500   CONTINUE
      print*,S
      RETURN
      END
C
C
```

```
      [illegible]
      CONTINUE
      Z(K,J) = Z(K,K) * Z(K,J)
  90  CONTINUE
      RETURN
      END
C
C
C
C
C
C     DESCRIPTION OF PARAMETERS
C       B   -  INPUT INVERSE MATRICS
C       N   -  ORDER OF MATRICS
C       C   -  INPUT SUBMATRICS
C     ---------------------------------------
C     --   SUBROUTINE FOR FINDING THE      --
C     --   ESTIMATED COEFFICIENT VECTOR    --
C     --                                   --
C     ---------------------------------------
      SUBROUTINE EST(B,N,C,D,m)
C
C
      DIMENSION B(m,m),C(m),D(m+1)
C
C
      D(1) = 1.0
      DO 100 I= 1,N
            K = I + 1
            d(k) = 0.0
         DO 110 J = 1,N
            D(K) = D(K) + B(I,J)*C(J)
 110     CONTINUE
            D(K) = -1.0*D(K)
 100  CONTINUE
      RETURN
      END
C
C
C
C     SUBROUTINE FOR CALCULATING SIGMA
C
C
C     PARAMETER DESCRIPTION
C         E   -  ESTIMATE VECTOR
C         N   -  ORDER OF MATRICS C
C         CC  -   COVARIANCE MATRICS
C         O   -  OUTPUT SIGMA
C         Q   -  VARIABLE
C
C     ---------------------------------------
C     --                                   --
C     --                                   --
C     --                                   --
C     ---------------------------------------
      SUBROUTINE SIGMA(E,N,CC,o,Q,m)
C
      DIMENSION E(m),CC(m,m),C(m)
C
```

```
                B(I,J) = A(I,J)
20          CONTINUE
10        CONTINUE
          RETURN
          END
C     ------------------------------------------------------
C     --                                                  --
C     --                PRINT THE MATRIX                  --
C     --                                                  --
C     ------------------------------------------------------
          SUBROUTINE SHOW(A,L)
          DIMENSION A(L,L)
          PRINT*,' PRINTING THE MATRIX ...'
          DO 10 I = 1,L
             DO 20 J = 1,L
                PRINT*,' MATRIX(',I,',',J,') =',A(I,J)
20           CONTINUE
10        CONTINUE
          RETURN
          END
C     ------------------------------------------------------
C     --                                                  --
C     --      MULTIPLY TWO MATRICES TOGETHER              --
C     --                                                  --
C     ------------------------------------------------------
          SUBROUTINE MULTIPLY(A,B,C,L)
          DIMENSION A(L,L),B(L,L),C(L,L)
          PRINT*,' MULTIPLYING THE MATRICES ...'
          DO 10 K = 1,L
             DO 20 I = 1,L
                SUM = 0
                DO 30 J = 1,L
                   SUM = A(K,J) * B(J,I) + SUM
30              CONTINUE
                C(K,I) = SUM
20           CONTINUE
10        CONTINUE
          RETURN
          END
C     ------------------------------------------------------
C     --    MATRIX INVERSION ROUNTINE                     --
C     --      inverts Kth row of matrix Z                 --
C     --                                                  --
C     ------------------------------------------------------
          SUBROUTINE SWEEP(Z,K)
          DIMENSION Z(31,31)
          Z(K,K) = 1/Z(K,K)
          DO 10 I = 1,K-1
10        Z(I,K) = -Z(K,K)*Z(I,K)
          CONTINUE
          DO 20 I = K+1,8
20        Z(I,K) = -Z(K,K)*Z(I,K)
          CONTINUE
          DO 60 J = 1,8
          IF (J .EQ. K) GO TO 60
          DO 30 I = 1,K-1
30        Z(I,J) = Z(I,J) + Z(I,K) * Z(K,J)
          CONTINUE
```

```
      ) END OF DISTANCE
C
      NF = N - P
C
      RAM = .5*(log(sigr) + log(sigs)) - log(sigp)
      DISTAN(inc)  = -2 *(float(np)*ram
      NUM(INC) = INC
1000  CONTINUE
C
C       SORTING OF DISTAN(I)
C
      imax = 0.0
      imin = 0.0
      DO 300 I = 1,NTOT - 1
          DO 400 J = I+1,NTOT
             IF(distan(I)  GT  DISTAN(J)) GO TO 400
                 imax = distan(I)
                 distan(I) = distan(J)
                 distan(J) = imax
                 imIN = NUM(I)
                 NUM(I) = num(J)
                 num(J) = imin
400          CONTINUE
300   CONTINUE
C
C     print output with descending order
C
      PRINT*, 'DISTANCE           ', 'SEQUENCE NO       ', 'STARTING POINT'
      DO 500 I = 1,NTOT
         IFIRST = N*(NUM(I) - 1) + 1
         PRINT*, DISTAN(I),NUM(I), IFIRST
500   CONTINUE
      CALL SHOW(DR,P)
      CALL COPYM(DR,X,P)
      DO 666 I = 1,P
         CALL SWEEP(DR,I)
666   CONTINUE
      CALL MULTIPLY(DR,X,Y,L)
      CALL SHOW(Y,L)
C     ------ ---------- ---------- ----------
C     --                                    --
C     --        E N D   O F   M A I N       --
C     --                                    --
C     ---------- ---------- ---------- ------
      STOP
      END
C
C
C     ---- - - ---------- ---------- ----------
C     --                                    --
C     --      COPY A MATRIX INTO ANOTHER    --
C     --                                    --
C     ---- ---------- ---------- ----------
      SUBROUTINE COPYM(A,B,L)
      DIMENSION A(L,L),B(L,L)
      PRINT*,' COPYING THE MATRIX   '
```

```
C       FIND SUBMATRICES OF CR, CS, CP
C
C        PRINT*,' '
         II = 0
         JJ = 0
        DO 14 I = 1,P
              II = I + 1
               DDR(I) = CR(II,1)
               DDS(I) = CS(II,1)
               DDP(I) = CP(II,1)
               DO 15 J = 1,P
                  JJ = J + 1
               DR(I,J) = CR(II,JJ)
               DS(I,J) = CS(II,JJ)
               DP(I,J) = CP(II,JJ)
C               PRINT*, DR(I,J),ds(i,j),dp(i,j)
15           CONTINUE
C               print*,' '
C               PRINT*, DDR(I),DDS(I),DDP(I)
14      CONTINUE
C
C
C
C        print*,' single dimension vector'
C        print*,( sr(i),i=1,np)
C        print*,' '
C        PRINT*,(SS(I),I=1,NP)
C        print*,' '
C        PRINT*,(SP(I),I=1,NP)
C
C
C       FIND INVERSE MATRICS OF DR,DS,DP
C
C
C
C
555     DO 115 I = 1,P
           CALL SWEEP(DR,I)
           CALL SWEEP(DS,I)
           CALL SWEEP(DP,I)
115     CONTINUE
C
C
C
C
C       FIND ESTIMATE OF 'A' VECTOR
C
        CALL EST(DRR,P,DDR,ER,mp)
        CALL EST(DSS,P,DDS,ES,mp)
        CALL EST(DPP,P,DDP,EP,mp)
C
C
C        FIND SIGMAR,SIGMAS,SIGMAP
C
C
        LL = P + 1
        CALL SIGMA(ER,LL,CR,SIGR,Q,mp1)
        CALL SIGMA(ES,LL,CS,SIGS,Q,mp1)
```

```
C     TRANSFER FDATA TO DATA

          IFIRST = N*(INC - 1) + 1
          ILAST  = N*(INC - 1) + 2*N
          J = 0
          DO 11 I =  IFIRST , ILAST
             J = J + 1
             DATA(J) = FDATA(I)
11    CONTINUE
C
C     PRINT INPUT DATA

      NN = 2*N
C      PRINT*,  ' '
C      PRINT*,  ' '
C      PRINT*, '-------------------------------------------------'
C      PRINT*,  ' '
C      PRINT*, INC, 'TH INPUT DATA'
C      PRINT*,  ' '
C      PRINT*, 'R(T)', (DATA(I), I=1, N)
C      PRINT*,  ' '
C      PRINT*,  S(T)', (DATA(I), I=N+1, NN)
C      PRINT*,  ' '
C
C
C         CALCULATE CR , CS, CP
C
C
C     LL : ORDER OF MATRICS CR, CS, CP
C
      LL = P + 1
      KK = 0
      DO 22 I = 1,LL
         DO 12 J = 1, LL
            SUM1 = 0.0
            SUM2 = 0.0
            DO 13 K = LL+1 ,N+ 1
               SUM1 = SUM1 + DATA(K-I)*DATA(K-J)
                KK = K + N
               SUM2 = SUM2 + DATA(KK-I)*DATA(KK-J)
13          CONTINUE
               CR(I,J) = sum1/float((N - P))
               CS(I,J) = SUM2 /float((N - P))
               CP(I,J) = ( CR(I,J) + CS(I,J) ) / 2
12       CONTINUE
22    CONTINUE
C
C     PRINT CR ,CS, CP
C
C      PRINT*,  '          COVARIANCE MATRICS OF R(T)'
C      PRINT*, (( CR(I,J), J=1,LL), I=1,LL)
C      PRINT*,  ' '
C      PRINT*,  '          COVARIANCE MATRICS OF S(T)'
C      PRINT*, (( CS(I,J), J=1,LL), I=1,LL)
C      PRINT*,  ' '
C      PRINT*,  '          COMBINED COVARIANCE MATRICS'
C      PRINT*, (( CP(I,J), J=1,LL), I=1,LL)
```

```
C      SET THE ORDER
C
       print*, 'order ='
       accept 200, p
C
C      READ DATA SET NAME
C
       read(5,7) ch
7      format(a5)
       print*, 'DATA SET NAME  = ', ch
C
C      PRINT WINDOW SIZE AND ORDER
C
       PRINT*,
       PRINT*, ' WINDOW SIZE = ', N
       PRINT*,
       PRINT*, ' ORDER OF AUTOREGRESSIVE FUNCTION = ', P
C
C
       WRITE(7,112) '   WINDOW SIZE = '
112    FORMAT(A15)
C
C
C      READ EKG DATA
C
C
       sum = 0.0
       ave = 0.0
       I = 0
8      I = I + 1
       READ(5,9,END =10 ) IDATA(I)
9      FORMAT(I7)
       sum = sum + idata(i)
       rdata(i) = idata(i)
       GO TO 8
C
C      PRINT TOTAL INPUT DATA
C
10     ITOT = I - 1
c       find average
       ave = float(sum) / float(itot)
c       print*, 'sum =', sum
c       print*, 'ave =', ave
c      substract average from all data
       do 9000 i = 1,itot
          rdata(i) = rdata(i) - ave
9000   continue
       PRINT*, 'TOTAL DATA NUMBER = ', ITOT
       PRINT*,
C
C
C
C      TOTAL EXAMINE SEQUENCE NUMBER = NTOT
C
C
       NTOT = ITOT/N -1
C
C      FIND DISTANCE BETWEEN SEQUENCE UNTIL NTOT SEQUENCE
C
```

```
C     DESCRIPTION OF PARAMETER
C          IDATA - INPUT DATA
C          DATA - DATA OF LENGTH 2*N
C          N    - WINDOW LENGTH
C          P    - ORDER OF AUTOREGRESSIVE FUNCTION
C          CR   - COVARIANCE MATRICS OF R(T)
C          CS   - COVARIANCE MATRICS OF S(T)
C          CP   - COMBINED MATRICS OF R(T) AND S(T)
C          DR,DS,DP,DDR,DDS,DDP - SUBMATRICS OF R(T) AND S(T)
C          ER,ES,EP - ESTIMATE MATRICS
C          SIGR,SIGS,SIGP - VARIANCE
C          RAM -
C          DISTAN -  FINAL OUTPUT OF THIS PROGRAM
C          INC -
C          IFIRST - FIRST INDEX OF EXAMINE SEQUENCE
C          ILAST - LAST INDEX OF EXAMINE SEQUENCE
C          CH - NAME OF DATA FILE
C          ITOT - TOTAL DATA NUMBER
C          NTOT - TOTAL EXAMINE SEQUENCE NUMBER
C          sr,ss,sp  - single dimension vector
C          mp - order of dr,ds,dp,ddr,dds,ddp
C          mp1 - order of cr,cs,cp,er,es,ep
C          mtd - order of sr,ss,sp
C
C
      REAL CR(31,31),CS(31,31),CP(31,31),DR(30,30)
     *     ,er(31),es(31)
     *    ,EP(31),DS(30,30),DP(30,30),DDR(30),
     *     DDS(30),DDP(30),D,SIGR,
     *     SIGS,SIGP,Q(31),sr(900),ss(900)
     *     ,sp(900),dpp(30,30),sum1,sum2
     *    ,RAM DISTAN(10000),imax,imin,
     *     ave,drr(30,30),dss(30,30),data(10000),rdata(10000)
     *    ,X(30,30),Y(30,30)
      INTEGER N,P,IDATA(10000),KK,LL,sum
     *       ,M(30),L(30) , NUM(10000),mp,mp1
      character*5 ch
      CHARACTER fname*12
      PRINT*,' Enter the filename '
      READ(*,111) fname
111   FORMAT(A12)
C
C          READ DATA AND ASSIGN WINDOW SIZE
C
C
      OPEN( UNIT = 5, FILE = fname , STATUS = 'OLD')
C
C
      mp = 30
      mp1 = 31
C
C     SET UP WINDOW SIZE
C
      PRINT*, WINDOW SIZE N ='
      ACCEPT 200,  n
200   FORMAT(I3)
```

```
FFFFF    AAA    CCCC  L        000   EEEEE  W   W
F       A   A  C      L       0   0  E      W   W
F       A   A  C      L       0   0  E      W   W
FFFF    A   A  C      L       0   0  EEEE   W   W
F       AAAAA  C      L       0   0  E      W W W
F       A   A  C      L       0   0  E      WW WW
F       A   A   CCCC  LLLLL    000   EEEEE  W   W
```

```
      000000     RRRRRRRR
      000000     RRRRRRRR
    00      00   RR     RR
    00      00   RR     RR
    00      00   RR     RR
    00      00   RR     RR
    00      00   RRRRRRRR
    00      00   RRRRRRR
    00      00   RR   RR
    00      00   RR   RR
    00      00   RR    RR
    00      00   RR    RR
      000000     RR     RR
      000000     RR     RR
```

```
FFFFFFFFFF     000000     RRRRRRRR
FFFFFFFFFF     000000     RRRRRRRR
FF           00      00   RR     RR
FF           00      00   RR     RR
FF           00      00   RR     RR
FF           00      00   RR     RR
FFFFFFFF     00      00   RRRRRRRR
FFFFFFFF     00      00   RRRRRRR
FF           00      00   RR   RR
FF           00      00   RR   RR
FF           00      00   RR    RR
FF           00      00   RR    RR
FF             000000     RR     RR
FF             000000     RR     RR
```

```
    44    44    44    44
    44    44    44    44
    44    44    44    44
    44    44    44    44
    44    44    44    44
    44    44    44    44
4444444444  4444444444
4444444444  4444444444
      44          44
      44          44
      44          44
      44          44
      44          44
      44          44
```

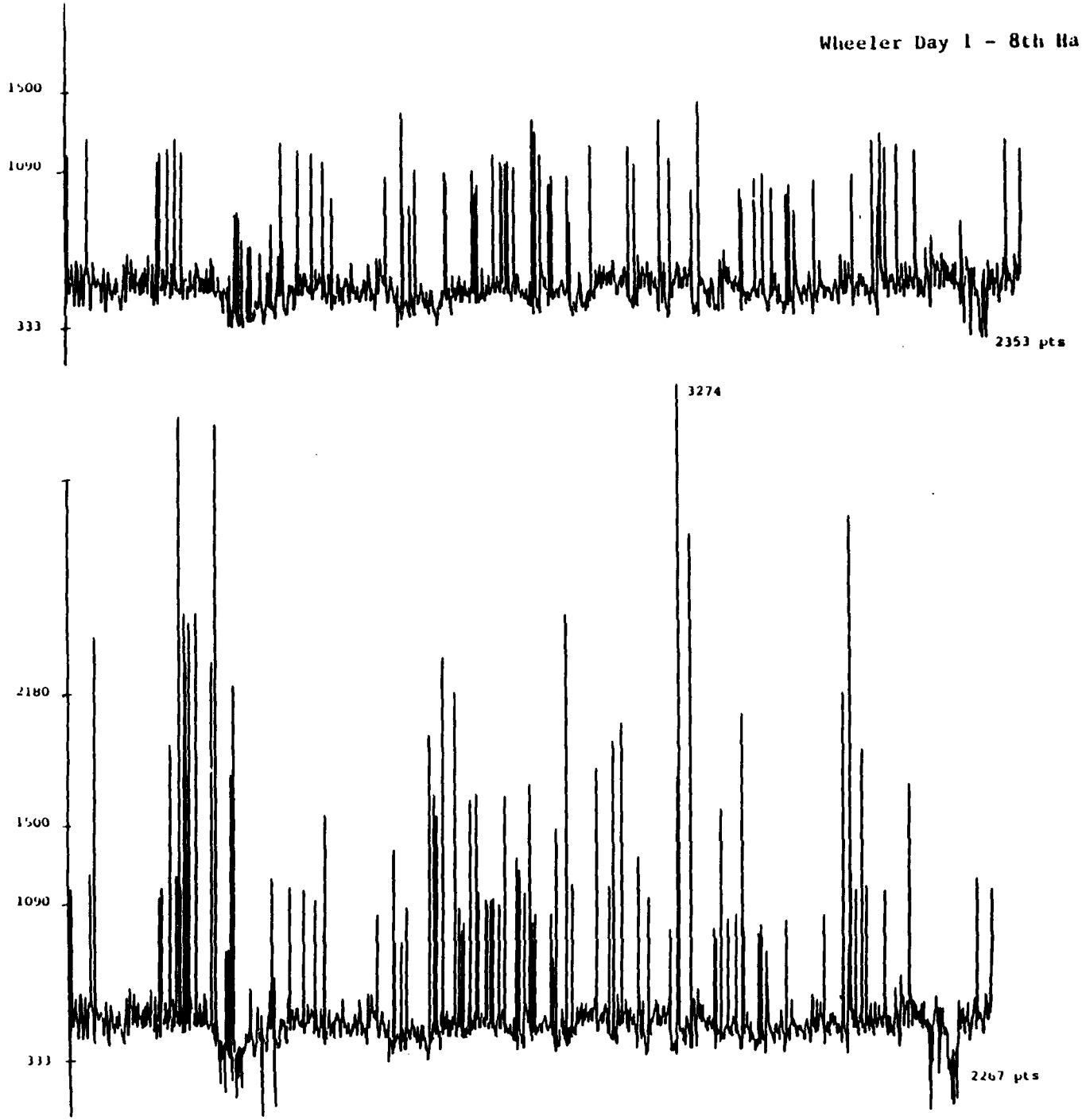
File _DUA1 [FACLOEW.ARMYJOB.FOR;44 (2708,19,0), last revised on 27-AUG-1986 09:55, is a 24 block sequential file owned by UIC [EECS,FACLOEW] The records are variable length with implied (CR) carriage control The longest record is 72 bytes.

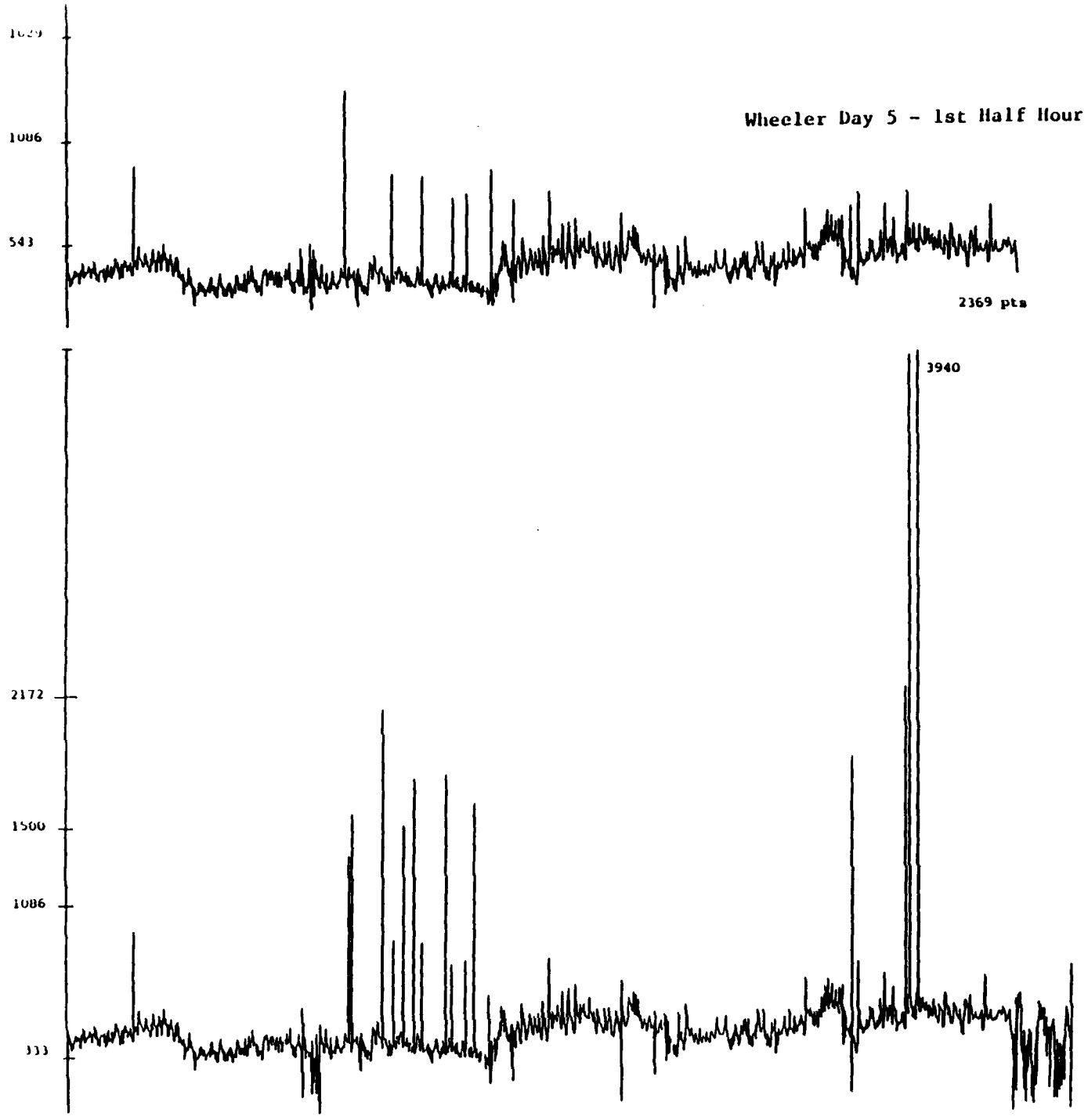
Job OR (1564) queued to SYS$PRINT on 27-AUG-1986 09:55 by user FACLOEW, UIC [EECS,FACLOEW], under account EECS at priority 100
started on printer _LPA0 on 27-AUG-1986 09:55 from queue LPA0

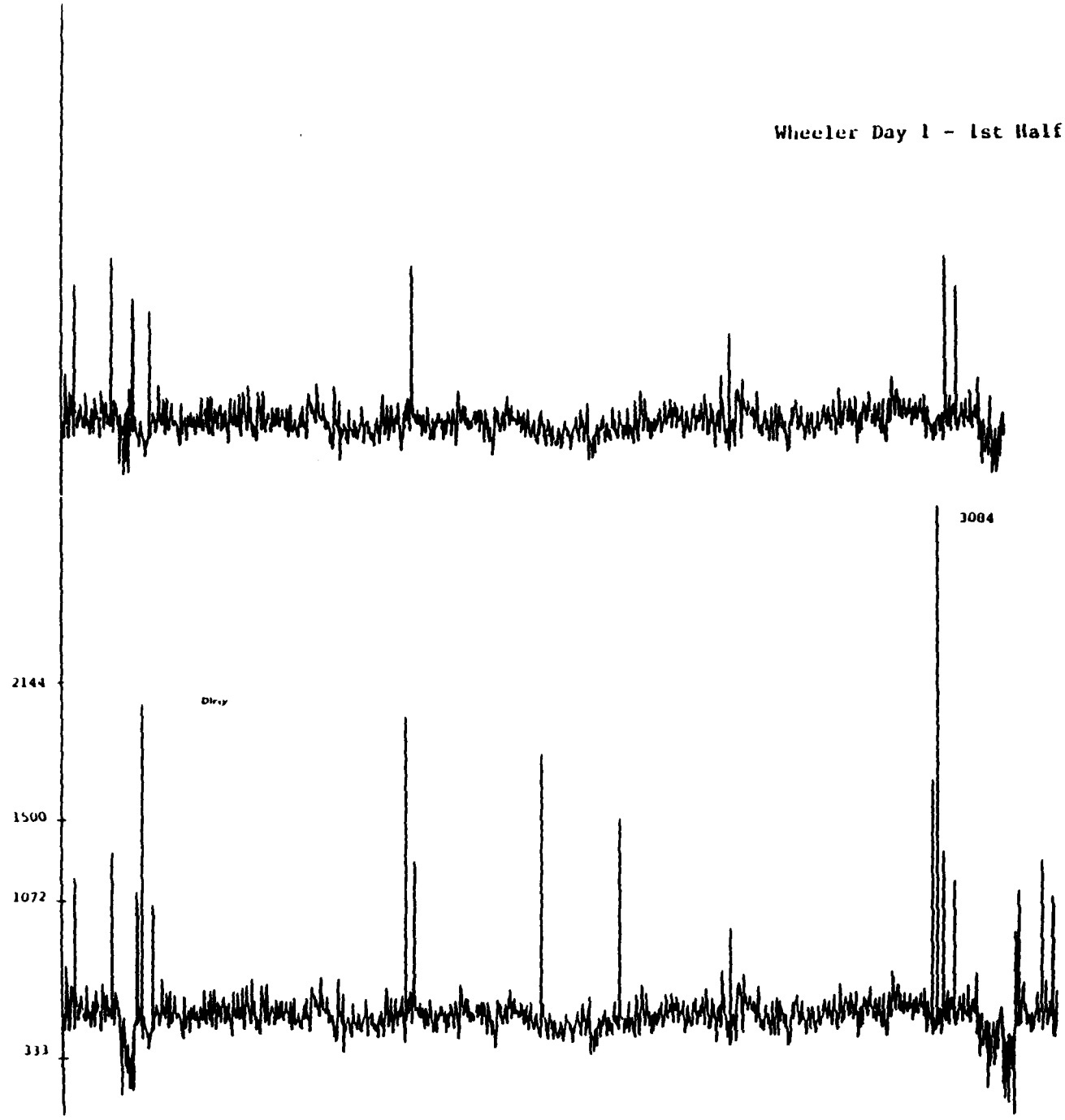
Digital Equipment Corporation - VAX/VMS Version V4.4

APPENDIX D

Plots of ECG data before and after preprocessing and noise removal


Wheeler Day 1 - 8th Half Hour
1500
1090
333
2353 pts
3274
2180
1500
1090
333
2267 pts


1629
1086
543
Wheeler Day 5 - 1st Half Hour
2369 pts
3940
2172
1500
1086
333


Wheeler Day 1 - 1st Half Hour
3084
2144
1500
1072
333

APPENDIX E

Part one of segmentation program

```
Begin
     NumPumpPrimes  := 0;  CleanNum  := 0;  ScratchNum  := 0;
     SetUp (DirtySet, CleanSet, ScratchSet, DirtyNum);
     For Index  := 1 to DirtyNum DO
               If DirtySet[Index] <> 0 Then       (* Transfer raw data into *)
                 Begin                            (* ScratchSet.  DO NOT    *)
                   ScratchNum  := ScratchNum + 1; (* TRANSFER ZEROS!        *)
                   ScratchSet[ScratchNum] := DirtySet[Index]
                 End; (* If *)
     PreProcess(ScratchSet, ScratchNum);
     REPEAT
          PreSweepStats(ScratchSet, ScratchNum);
          SweepNumber := 0;
          InitStats;
          REPEAT
               NoChanges := True;
               SweepNumber := SweepNumber + 1;
               Sweep (ScratchSet, ScratchNum, CleanSet, CleanNum,
                                              NoChanges, NumBad, NumFinal);
               For Index := 1 to NumFinal DO
                    ScratchSet[Index] := CleanSet[Index]; { Transfer back }
               ScratchNum := NumFinal;
          UNTIL NoChanges;
          PrintStats;
          If NOT AllClean(ScratchSet, ScratchNum)
               Then PumpPrime(ScratchSet, ScratchNum);
     UNTIL AllClean(ScratchSet, ScratchNum);
     For Index  := 1 to ScratchNum DO
         CleanSet[Index]  := ScratchSet[Index];
     CleanNum  := ScratchNum;
     Writeln;
     Writeln(' The file has been entirely cleaned.');
     Writeln(' The final number of points is :  ', CleanNum);
     SetDown(CleanSet, CleanNum) ;
End   (* Clean *)
```

```
        Open ( FN,
               DataFile,
               History  = old),
        (Option for "My_File" is "My_File  = 'MUA0 : [] DirtyData Dat' " )
        ( which will read the test data on the tape organized like a DIR )
        RESET(FN) ;
        For Index  = 1 to 3500 do
             Begin
                  DirtySet[Index]  = 0;
                  CleanSet[Index]  = 0;
                  ScratchSet[Index]  = 0
             End. (* For *)
        DirtyNum  = 0;
        While NOT EOF(FN) DO
             Begin
                  DirtyNum  = DirtyNum + 1;
                  ReadIn (FN, DirtySet[DirtyNum])
             End. (* While *)
        Close(FN)
End. (* SetUp *)



Procedure SetDown (Data : DataArray; Num : Integer);

VAR Index : Integer;
    FN : Text;
    OutFile   Varying [30] of CHAR;

Begin
     WriteIn (' Please specify the output file name ... ');
     ReadIn (OutFile);
     Open ( FN,
            OutFile,
            History  = new );
     ReWrite (FN) ;
     For Index  = 1 to Num Do
         WriteIn (FN, Data[Index] : 5);
     Close(FN)
End. (* SetDown *)


{-------------------------------------------------------------}
{--                                                         --}
{--            M A I N        P R O G R A M                 --}
{--            -------        -------------                 --}
{--                                                         --}
{-------------------------------------------------------------}
```

```
          Else (* point is good *)
             Begin
                 OutAry[Outdex]  := Ary[Index];
                 OutPoints  := OutPoints + 1;
                 Outdex  := Outdex + 1
             End. (* Else *)
          Index  := Index + 1;
     UNTIL Index >= NumPoints + 1;
     If ChangeMade Then
          Begin
               For Outdex := 1 To OutPoints DO
                    Data[Outdex] := OutAry[Outdex];
               Num := OutPoints
          End (* If *)
     Else PumpPrimeB(Data, Num);
     NumPumpPrimes := NumPumpPrimes + 1
End. (* PumpPrime *)



Function AllClean(VAR ScratchSet : DataArray; ScratchNum : Integer) : Boolean;

VAR  DirtFound   Boolean;
     Datum, Index : Integer;

Begin
     DirtFound  := False;
     Index := 1;
     While (Index <= ScratchNum) AND (NOT DirtFound) DO
       Begin
          If NOT Good(ScratchSet[Index]) Then DirtFound := True;
          Index  := Index + 1
       End. (* While *)
     AllClean  := NOT DirtFound
End. (* AllClean *)



Procedure SetUp(VAR  DirtySet, CleanSet, ScratchSet : DataArray;
                                                  VAR DirtyNum : Integer);
VAR Index   Integer,
    FN : Text,
    DataFile   Varying [30] of CHAR;

Begin
     Writeln (' Please specify the input file name ...');
     Readln (DataFile);
```

```
                Begin
                   Ary[IndBad]  := NewValue;
                   For Index  := (IndBad + 1) To (NumPoints - 1) DO
                      Ary[Index]  := Ary[Index + 1]
                End; (* Else *)
             Num  := NumPoints - 1;
             For Index  := 1 To Num DO
                Data[Index]  := Ary[Index]
          End (* Else *)
End; (* PumpPrime8 *)
```

```
Procedure PumpPrime(VAR Data : DataArray; VAR Num : Integer);

VAR  ChangeMade   Boolean;
     Ary, OutAry   DataArray;
     Index, Outdex, OutPoints, NumPoints : Integer;

Begin
     Writeln; Writeln(' Pump-Primer (A) activated. '); Writeln;
     OutPoints := 0;  NumPoints := Num;
     For Index := 1 To NumPoints DO
          Ary[Index] := Data[Index];
     Index := 1;  ChangeMade := False;  Outdex := 1;
     REPEAT
          While Good(Ary[Index]) AND (Index < NumPoints) DO
             Begin
               OutAry[Outdex] := Ary[Index];
               OutPoints := OutPoints + 1;
               Index := Index + 1;
               Outdex := Outdex + 1
             End; (* While *)
          If (NOT Good(Ary[Index])) AND
             ((GoodLeft10(Index, Ary)) OR (GoodRight10(Index,Ary,NumPoints))) Then
             Begin
              If Ary[Index] < 333 Then
                PumpPrimeSmallA(Index, Ary, Outdex, OutAry, NumPoints,
                                                        OutPoints, ChangeMade)
              Else
                                          (* Bad point is >= 1500 *)
                PumpPrimeLargeA(Index, Ary, Outdex, OutAry, NumPoints,
             End (* If *)                                  OutPoints, ChangeMade)
          Else If NOT Good(Ary[Index]) Then
             Begin
                 Writeln(' Bad point does not have 10 good neighbors on ',
                                                        'either side. ');
                 Writeln(' (Pump-Primer A cannot fix this point)');
                 OutAry[Outdex] := Ary[Index];   (* Pass bad value *)
                 Outdex := Outdex + 1;
                 OutPoints := OutPoints + 1
```

```
        Writeln(' see WARNING !!!  LONGEST GOOD RUN SHORTER THAN 10 LONG !')
End. (* FindLongestRun *)



Procedure PumpPrimeB(VAR Data : DataArray; VAR Num : Integer);

VAR  Index, NumPoints, IndLeftBad, IndRightBad, IndBad, NewValue : Integer;
     Ary : DataArray;
     AddedToLeft : Boolean;

Begin
     Writeln(' Pump-Primer B activated.  (B MUST change a value)');
     NumPoints := Num;
     For Index := 1 to NumPoints DO
          Ary[Index] := Data[Index];
     FindLongestRun(Ary, NumPoints, IndLeftBad, IndRightBad);
     If IndLeftBad = 0 Then IndBad := IndRightBad
     Else IndBad := IndLeftBad;
     If Ary[IndBad] > 333 Then
        Begin
          Writeln(' ******* WARNING !!!  LARGE VALUE AS TARGET !!! ');
          Writeln(' *******          PROGRAM WILL NOT STOP !!!')
        End (* If *)
     Else
        Begin
           If IndBad = 1 Then
              Begin
                 NewValue := Ary[1]+Ary[2];
                 AddedToLeft := False
              End (* If *)
           Else If IndBad = NumPoints Then
              Begin
                 NewValue := Ary[NumPoints-1] + Ary[NumPoints];
                 AddedToLeft := True;
              End (* Else If *)
           Else If Ary[IndBad-1] <= Ary[IndBad+1] Then
              Begin
                 NewValue := Ary[IndBad-1] + Ary[IndBad];
                 AddedToLeft := True
              End (* Else If *)
           Else
              Begin
                 NewValue := Ary[IndBad] + Ary[IndBad+1];
                 AddedToLeft := False
              End; (* Else *)
           If AddedToLeft Then
              Begin
                 Ary[IndBad-1] := NewValue;
                 For Index := IndBad To (NumPoints - 1) DO
                    Ary[Index] := Ary[Index + 1]
              End (* If *)
```

```
            End (* While *)
        Else if Accum > Ary[Index] Then
          While Accum > Ary[Index] DO    (* Distribute - residual *)
            Begin
                 NewValues[Counter]  = NewValues[Counter] - 1;
                 Accum  = Accum - 1;
                 Counter  = Counter - 1;
                 If Counter = 0 Then Counter := NumPartit
            End. (* While *)
        If SmallerMean = LeftMean then
         For Counter  = 1 To NumPartit DO
           Begin
             OutAry[Outdex]  = NewValues[Counter];
             OutPoints  = OutPoints + 1;
             Outdex  = Outdex + 1
           End (* For *)
        Else
         For Counter  = NumPartit DownTo 1 DO
           Begin
             OutAry[Outdex]  = NewValues[Counter];
             OutPoints  = OutPoints + 1;
             Outdex  = Outdex + 1
           End. (* For *)
        ChangeMade  = True;
        Writeln(' Pump-Primer A succeeds in fixing medium or large value. ')
End. (* PumpPrimeLargeA *)
```

```
Procedure FindLongestRun(Ary : DataArray; NumPoints : Integer;
                         VAR IndLeftBad, IndRightBad : Integer);

VAR  Index, TempIndLeftBad, TempIndRightBad, Longest : Integer;

Begin
     TempIndLeftBad  = 0;  TempIndRightBad  = 1;  Longest  = 0;
     Index  = 0;
     REPEAT
          Index  = Index + 1;
          If Good(Ary[Index]) Then
             TempIndRightBad  = TempIndRightBad + 1;
          If (NOT Good(Ary[Index])) OR (Index = NumPoints) Then
             Begin
                If (TempIndRightBad - TempIndLeftBad) > Longest Then
                   Begin
                      Longest  = TempIndRightBad-TempIndLeftBad;
                      IndRightBad  = TempIndRightBad;
                      IndLeftBad  = TempIndLeftBad
                   End; (* If *)
                TempIndLeftBad  = Index;
                TempIndRightBad  = Index + 1
             End. (* If *)
```

```
Procedure PumpPrimeLargeAt(Index  : Integer; VAR Ary  : DataArray;
                           VAR OutIdx  : Integer; VAR OutAry : DataArray;
                           NumPoints : Integer;VAR OutPoints : Integer;
                           VAR ChangeMade : Boolean);

VAR  NearMedian, FiveMean, OtherSideMean, FarMean, LeftMean, RightMean,
     NumPartit, ComDiff, SmallerMean, Counter, Accum        : Integer;
     NewValues   Array[1  100] of Integer;
     RN   Real;

Begin
     If GoodLeft10(Index, Ary) Then
          Begin
               NearMedian  := GetMedian3(Ary, Index-3);
               FiveMean  := Mean5(Index-5, Ary);
               OtherSideMean := MeanOfGoodIn5(Index+1, Ary, NumPoints);
               If OtherSideMean <> 0 Then
                    FarMean := ((2*FiveMean)+OtherSideMean) DIV 3
               Else FarMean := FiveMean;
               LeftMean  := NearMedian;
               RightMean := FarMean
          End (* If *)
     Else
          Begin
               NearMedian := GetMedian3(Ary, Index+1);
               FiveMean  := Mean5(Index+1, Ary);
               OtherSideMean := MeanOfGoodIn5(Index-5, Ary, NumPoints);
               If OtherSideMean <> 0 Then
                    FarMean := ((2*FiveMean)+OtherSideMean) DIV 3
               Else FarMean := FiveMean;
               LeftMean := FarMean;
               RightMean  := NearMedian
          End; (* Else If *)
     RN := (2.0 * Ary[Index]) / (LeftMean+RightMean);
     If RN <= 1.3 Then
          Begin
               Writeln('!!!!!!!!!!!!! RN in PumpPrimer <= 1.3 !!!!');
               Writeln('NumPartit set to 2');
               NumPartit  := 2
          End (* If *)
     Else If RN < 2.0 Then NumPartit  := 2
     Else NumPartit  := Round(RN);
     ComDiff  := Round(ABS((1.0*(LeftMean - RightMean)) / (NumPartit - 1)));
     If LeftMean < RightMean Then SmallerMean := LeftMean
     Else SmallerMean := RightMean;
     Accum  := 0;
     For Counter  := 1 to NumPartit DO     (* Partition via arith prog *)
          Begin
               NewValues[Counter] := SmallerMean+((Counter-1) * ComDiff);
               Accum := Accum + SmallerMean + ((Counter-1) * ComDiff)
          End; (* For *)
     Counter  := NumPartit;
     If Accum < Ary[Index] Then
       While Accum < Ary[Index] DO     (* Distribute + residual *)
          Begin
               NewValues[Counter] := NewValues[Counter] + 1;
               Accum := Accum + 1;
               Counter := Counter - 1;
```

```
                    OutAry[Outdex-1]  := NewValue;
                    Writeln;
                    Writeln(' Pump-Primer A succeeds in changing small value ');
                    Writeln(' Value added to left neighbor. '); Writeln;
               End (* If *)
          Else If Success AND (AddedSideLeft = False) Then
               Begin
                    OutAry[Outdex]  := NewValue;
                    Outdex := Outdex + 1;
                    Index := Index + 1;  (* skip a value on Ary *)
                    OutPoints  := OutPoints + 1;
                    Writeln;
                    Writeln(' Pump-Primer A succeeds in fixing small value. ');
                    Writeln(' Value added to right neighbor. '); Writeln
               End;  (* Else If *)
          If Success Then ChangeMade := True
          Else Begin                              (* pass bad value *)
                    OutAry[Outdex] := Ary[Index];
                    Outdex := Outdex + 1;
                    OutPoints := OutPoints + 1
               End (* Else *)
     End.  (* PumpPrimeSmallA *)
```

```
Function MeanOfGoodIn5(Index : Integer; Ary : DataArray; NumPoints : Integer)
                                                                        : Integer;

(* Index must be the index of the FIRST of the 5.  OK to give EOF, etc. *)

VAR  Sum, Num, Counter : Integer;

Begin
     Sum := 0;  Num := 0;
     For Counter  := 0 to 4 DO
          If ((Index+Counter)>0) AND ((Index+Counter)<=NumPoints) Then
               If Good(Ary[Index+Counter]) Then
                    Begin
                         Sum := Sum + Ary[Index + Counter];
                         Num := Num + 1
                    End; (* If *)
     If Num <> 0 Then MeanOfGoodIn5 := Sum DIV Num
     Else MeanOfGoodIn5 := 0
End;  (* MeanOfGoodIn5 *)
```

```
                                  VAR Index    : Integer; VAR Ary    : DataArray;
                                  VAR Untdex   : Integer; VAR OutAry : DataArray;
                                  NumPoints    : Integer;VAR OutPoints : Integer;
                                  VAR ChangeMade : Boolean));

VAR  NewValue, Counter, LeftSum, RightSum, LeftMean, RightMean : Integer;
     AddedSideLeft, Success : Boolean;
{* CONST  AllowedDev = 0.30;  DEACTIVATED 3/14/86 *}
Begin
     AddedSideLeft := True;  Success := False;
     If (GoodLeftIO(Index,Ary) AND GoodRightIO(Index,Ary,NumPoints))
        AND ((NOT (Index = 1)) AND (NOT (Index = NumPoints))) Then
          Begin
               LeftSum  := Ary[Index] + Ary[Index-1];
               RightSum  := Ary[Index] + Ary[Index+1];
               LeftMean  := MeanLeftIO(Index,Ary);
               RightMean := MeanRightIO(Index,Ary);
               If ABS(LeftSum-LeftMean) > ABS(RightSum-RightMean) Then
                    Begin
                         NewValue := RightSum;
                         AddedSideLeft := False
                    End {* If *}
               Else NewValue := LeftSum;
               If NewValue < 1500 Then Success := True
          End {* If *}
     Else If GoodLeftIO(Index,Ary)
          AND ((NOT (Index = 1)) AND (NOT (Index = NumPoints))) Then
          Begin
               LeftSum := Ary[Index]+Ary[Index-1];
               LeftMean := MeanLeftIO(Index,Ary);
               If (LeftSum < 1500)
{* AND
  (ABS(LeftMean-LeftSum) < (AllowedDevsLeftMean)) DEACTIVATED 3/14/86 *}
                 Then
                   Begin
                        NewValue := LeftSum;
                        Success := True
                   End {* If *}
               End {* Else If *}
     Else If GoodRightIO(Index,Ary,NumPoints)
          AND ((NOT (Index = 1)) AND (NOT (Index = NumPoints))) Then
          Begin
               RightSum := Ary[Index]+Ary[Index-1];
               RightMean := MeanRightIO(Index,Ary);
               If (RightSum < 1500)
{* AND
(ABS(RightMean-RightSum) < (AllowedDevsRightMean))  DEACTIVATED 3/14/86 *}
                 Then
                  Begin
                       NewValue := RightSum;
                       AddedSideLeft := False;
                       Success := True
                  End {* If *}
          End; {* Else If *}
     If NOT Success Then
        Writeln(' Small point not fixable by Pump Primer A');
     If Success AND (AddedSideLeft = True) Then
```

```
Function Mean5(Index : Integer; Ary : DataArray) : Integer;

{* Index must be the index of the FIRST of the 5 *}

Begin
     Mean5  = (Ary[Index]+Ary[Index+1]+Ary[Index+2]+Ary[Index+3]+
               Ary[Index+4]) DIV 5
End. {* Mean5 *}



Function Mean10(Index : Integer; Ary : DataArray) : Integer;

{* Index must be the index of the FIRST of the 10. *}

Begin
     Mean10  = (Ary[Index]+Ary[Index+1]+Ary[Index+2]+Ary[Index+3]+
               Ary[Index+4]+Ary[Index+5]+Ary[Index+6]+Ary[Index+7]+
               Ary[Index+8]+Ary[Index+9]) DIV 10
End. {* Mean10 *}



Function MeanLeft10(Index : Integer; Ary : DataArray) : Integer;
Begin
     MeanLeft10  = Mean10(Index-10,Ary)
End.



Function MeanRight10(Index : Integer; Ary : DataArray) : Integer;
Begin
     MeanRight10  = Mean10(Index+1,Ary)
End.
```

```
Begin
     NumSmall  := 0;  NumMedium  := 0;  NumLarge  := 0;
     SumSmall  := 0;  SumMedium  := 0;  SumLarge  := 0;
     For Index  := 1 to NumPoints DO
          Begin
               If Data[Index] < 333 Then
                    Begin
                         NumSmall  := NumSmall + 1;
                         SumSmall  := SumSmall + Data[Index]
                    End
               Else If Data[Index] > 2999 Then
                    Begin
                         NumLarge  := NumLarge + 1;
                         SumLarge  := SumLarge + Data[Index]
                    End
               Else If Data[Index] > 1499 Then
                    Begin
                         NumMedium := NumMedium + 1;
                         SumMedium := SumMedium + Data[Index]
                    End
          End; (* For *)
     Writeln;
     If NumPumpPrimes = 0 Then Writeln(' Post-Preprocessor Statistics :')
     Else Writeln(' Statistics for file after ',NumPumpPrimes,' Pump-Primes:');
     Writeln;
     Writeln(' Number of Small Points  = ',NumSmall :4,'     Sum = ',
                                                    SumSmall :11:0);
     Writeln(' Number of Medium Point  = ',NumMedium :4,'     Sum = ',
                                                    SumMedium :11:0);
     Writeln(' Number of Large Points  = ',NumLarge :4,'     Sum = ',
                                                    SumLarge :11:0);
     Writeln;
     Writeln(' The following are the lengths (in numbers of data points) ');
     Writeln(' of all of the runs of good points:');
     Goods := 0;  NumGoodRuns := 0;
     For Index  := 1 to NumPoints DO
          Begin
               If Good(Data[Index]) Then Goods := Goods + 1;
               If ((NOT Good(Data[Index])) OR (Index = NumPoints))
                  AND (Goods <> 0) Then
                    Begin
                         Writeln(' ',Goods);
                         NumGoodRuns := NumGoodRuns + 1;
                         Goods := 0
                    End (* If *)
          End; (* For *)
     Writeln; Writeln(' There are ',NumGoodRuns,' good runs.');
     Writeln
End; (* PreSweepStats *)
```

```
Procedure PreProcess(VAR Data: DataArray; VAR NumPoints : Integer);

VAR
      Index, NewNumPoints,
      NumCut   Integer,
      SumCut   Real;

Begin
      Index  = NumPoints;
      While (NOT SmallIn4of5WindowsOf5(Data, Index))
             AND (Index >= (2*(NumPoints DIV 3))) DO
                 Index := Index - 1;
      If Index > (2*(NumPoints DIV 3)) Then  (* Bad section was found *)
         Begin
             Index  := Index - 150;  (* *** THIS IS THE AMOUNT JUMPED BACK *** *)
             If Index < 1 Then Index := 1;  (* Safety feature *)
             While SmallsInLeft15(Data, Index) AND (Index > 15) DO
                 Index := Index - 15;
             While (NOT ThreeSmallsInRight15(Data, Index, NumPoints))
                   AND (Index < NumPoints) DO
                 Index := Index + 1;
             While Data[Index] >= 333 DO
                 Index := Index + 1;
             Index := Index - 1;
             NewNumPoints := Index;
             NumCut := NumPoints - NewNumPoints;
             SumCut := 0;
             For Index := (NewNumPoints+1) TO NumPoints DO
                 SumCut := SumCut + Data[Index];
             Writeln;
             Writeln(' @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@');
             Writeln(' PREPROCESSOR HAS RUN. ');
             Writeln(' Number of values cut from end of file = ', NumCut);
             Writeln(' Sum of values cut from end of file = ', SumCut :11:0);
             Writeln(' @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@');
             Writeln;
             NumPoints := NewNumPoints
         End (* If *)
End. (* PreProcess *)




Procedure PreSweepStats(VAR Data : DataArray; VAR NumPoints: Integer);

VAR   NumSmall, NumMedium, NumLarge, NumGoodRuns, Index,
      Goods   Integer;
      SumSmall, SumMedium, SumLarge : Real;
```

```
Function SmallIn4Of5WindowsOf5(Ary : DataArray; Index : Integer) : Boolean;
Var s1,s2,s3,s4,s5 : Integer;
Begin   (* Includes Index Point *)
     s1  := OneIfSmallsIn5(Ary,Index);
     s2  := OneIfSmallsIn5(Ary,Index-5);
     s3  := OneIfSmallsIn5(Ary,Index-10);
     s4  := OneIfSmallsIn5(Ary,Index-15);
     s5  := OneIfSmallsIn5(Ary,Index-20);
     SmallIn4Of5WindowsOf5 := (s1+s2+s3+s4+s5) >= 4
End.  (* SmallIn4Of5WindowsOf5 *)
```

```
Function SmallsInLeft15(Ary : DataArray; Index : Integer) : Boolean;
Var Temp : Boolean; i : Integer;
Begin   (* Does NOT include Index Point *)
     Temp := False;
     For i := 1 To 15 DO
         If (Index - i) > 0 Then
             If Ary[Index - i] < 333 Then
                 Temp := True;
     SmallsInLeft15 := Temp
End.  (* SmallsInLeft15 *)
```

```
Function ThreeSmallsInRight15(Ary : DataArray; Index, NumPoints : Integer)
                                                                  : Boolean;
Var   i, Smalls : Integer;
Begin   (* Includes Index Point *)
     Smalls  := 0;
     For i   := 0 To 14 DO
          If (Index + i) <= NumPoints Then
              If Ary[Index + i] < 333 Then
                  Smalls := Smalls + 1;
     ThreeSmallsInRight15 := Smalls >= 3
End.  (* ThreeSmallsInRight15 *)
```

```
      Writeln ('SWEEP # ', SweepNumber, ' IN PROGRESS     ',
                           NumBad, ' INITIAL BAD VALUES');  Writeln;
      OutPointer  := 1; InPointer  := 0;
      While InPointer < NumPoints DO                          (* Cleaning Loop *)
           Begin
             InPointer  := InPointer + 1;
             If InArray[InPointer] < 333 Then
               Begin
                SmallCase(InArray, OutArray, InPointer, OutPointer, NumPoints,
                                                    NoChanges, SkipOne);
                If SkipOne Then InPointer := InPointer + 1
               End (* If *)
             Else If InArray[InPointer] > 2999 Then
                LargeCase(InArray, OutArray, InPointer, OutPointer, NumPoints,
                                                              NoChanges)
             Else If InArray[InPointer] > 1499 Then
                MediumCase(InArray, OutArray, InPointer, OutPointer, NumPoints,
                                                              NoChanges)
             Else (* Point is good *)
                Begin
                   OutArray[OutPointer] := InArray[InPointer];
                   OutPointer := OutPointer + 1
                End (* Else *)
           End; (* While *)
      NumBad  := 0;
      For Index  := 1 to (OutPointer - 1) DO              (* Send to Array and *)
           Begin                                          (* count bad points *)
                OutData[Index] := OutArray[Index];
                If (NOT Good(OutArray[Index])) Then
                     NumBad := NumBad + 1
           End; (* For *)
      NumFinal  := OutPointer - 1;
      OutNum  := OutPointer - 1;
      Write (' SWEEP COMPLETED..  ');
      If NoChanges Then Writeln ('NO CHANGES MADE THIS SWEEP.')
      Else Writeln (NumBad, ' VALUES ARE STILL BAD.');
      Writeln; Writeln
End; (* Sweep *)
```

```
Function OneIfSmallsIn3(Ary : DataArray; Index : Integer) : Integer;
Var Temp, i   Integer;
Begin  (* Includes Index Point *)
      Temp  := 0;
      For i  := 0 To 4 DO
         If (Index - i) > 0 Then
            If Ary[Index - i] < 333 Then
               Temp := 1;
      OneIfSmallsIn3 := Temp
End; (* OneIfSmallsIn3 *)
```

```
                    Accum  = Accum - 1;
                    Index  = Index - 1;
                    If Index = 0 Then Index := NumPartit
                End; (* While *)

            For Index  = 1 to NumPartit DO          (* Send out new values *)
                Begin
                  If SmallerMean = LeftMean Then
                    OutArray[OutPointer] := NewValues[Index]
                  Else
                    OutArray[OutPointer] := NewValues[NumPartit+1-Index];

                  OutPointer := OutPointer + 1
                End; (* For *)
            NoChanges := False;
            Writeln(' Algorithm Succeeds!');
            Writeln
        End (* If *)
    Else
        Begin
            OutArray[OutPointer] := InArray[InPointer];
            OutPointer := OutPointer + 1;
            Writeln (' Algorithm Fails!');
            Writeln
        End (* Else *)
End; (* LargeCase *)
```

```
Procedure Sweep(VAR InData: DataArray; VAR InNum : Integer;
                VAR OutData : DataArray; VAR OutNum : Integer;
                VAR NoChanges : Boolean;  VAR NumBad, NumFinal : Integer);

(* This is the top-level data-cleaning procedure - it "sweeps" through      *)
(* the data file once, replacing "bad" values with better values, and       *)
(* returning the new data in an Array (OutData).  The flag "NoChanges"      *)
(* is returned as True if no changes were made to the data.                 *)
(* GLOBAL VARIABLE ACCESSED : SweepNumber                                   *)
                                                                            *)
VAR  NumPoints, InPointer, OutPointer, Index : Integer;
     InArray, OutArray : DataArray;
     SkipOne   Boolean;

Begin
    NumBad  := 0;   NumPoints := 0;
    While NumPoints <> InNum DO
        Begin                                       (* Fill InArray and *)
            NumPoints := NumPoints + 1;             (* count bad points *)
            InArray[NumPoints] := InData[NumPoints];
            If (NOT Good(InArray[NumPoints])) Then
                NumBad := NumBad + 1
```

```
          Writeln(' LeftSum at failure = ', LeftSum);
          Writeln(' RightSum at failure = ', RightSum)
     End; (* Else *)



If Success Then
     Begin
          If InArray[InPointer] < 16000 Then
               StatAryInd := (InArray[InPointer]*2) DIV 1000
          Else
               StatAryInd  := 32;

          LFixed[SweepNumber,StatAryInd] :=
               LFixed[SweepNumber,StatAryInd]+1;

          RN  := (2.0 * InArray[InPointer]) / (LeftMean+RightMean);

          If RN <= 1.3 Then
               Begin
                    Writeln ('!!!!!!!!!!! RN <= 1.3 !!!!!!!!!!!!');
                    Writeln ('NumPartit set to 2');
                    NumPartit := 2;
               End (* If *)
          Else
               IF RN < 2.0 then
                    NumPartit := 2
               Else
                    NumPartit := Round(RN);

          ComDiff :=
               Round(ABS((1.0*(LeftMean-RightMean))/(NumPartit-1)));

          If LeftMean < RightMean Then
               SmallerMean := LeftMean
          Else
               SmallerMean  := RightMean;

          Accum := 0;

          For Index  := 1 to NumPartit DO     (* Partition via arith prog *)
               Begin
                    NewValues[Index] := SmallerMean+((Index-1) * ComDiff);
                    Accum := Accum + SmallerMean + ((Index-1) * ComDiff)
               End; (* For *)

          Index := NumPartit;
          If Accum < InArray[InPointer] Then
            While Accum < InArray[InPointer] DO ( Distribute+residual )
               Begin
                    NewValues[Index] := NewValues[Index] + 1;
                    Accum := Accum + 1;
                    Index := Index - 1;
                    If Index = 0 Then Index := NumPartit
               End (* While *)
          Else
               If Accum > InArray[InPointer] Then
            While Accum > InArray[InPointer] DO ( Distribute-residual )
               Begin
```

APPENDIX F

Classitier Variable Definitions

The variables are:

J_{max} = number of components in the feature vector

N = number of observations

D_i = observation point in J_{max}-space

D(J) = array of the N observations ranked according to the Jth complement

R_i = a region in J_{max}-space

C_i = classification assigned to R_i

IN_i = rank of first point in D(j) which is contained in R_i

FN_i = rank of last point in D(J) which is contained in R_i

K = the size of the blocks into which the feature-value axis is initially partitioned

I_{max} = number of new regions formed

S(.) = stored value of .